MOTOROLA

Global Telecom Solution Sector

i325
Multi-Communication Device
J2ME™ Developers' Guide

Table of Contents

Document Overview

This guide describes the procedures used to develop a J2ME™ compliant application for the i325 multi-communication device. Also included is information on developing and packaging applications for installation as well as a step-by-step procedure for setting up a debug environment. Additionally, answers to frequently asked questions are addressed in this guide. Detailed information on the Java 2 Micro Edition environment is not provided.

Disclaimer

Motorola reserves the right to make changes without notice to any products or services described herein. "Typical" parameters, which may be provided in Motorola Data sheets and/or specifications can and do vary in different applications and actual performance may vary. Customer's technical experts must validate all "Typicals" for each customer application.

MOTOROLA MAKES NO WARRANTY WITH REGARD TO THE PRODUCTS OR SERVICES CONTAINED HEREIN. IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE GIVEN ONLY IF SPECIFICALLY REQUIRED BY APPLICABLE LAW. OTHERWISE, THEY ARE SPECIFICALLY EXCLUDED.

No warranty is made as to coverage, availability, or grade of service provided by the products or services, whether through a service provider or otherwise.

No warranty is made that the software will meet your requirements or will work in combination with any hardware or applications software products provided by third parties, that the operation of the software products will be uninterrupted or error free, or that all defects in the software products will be corrected.

IN NO EVENT SHALL MOTOROLA BE LIABLE, WHETHER IN CONTRACT OR TORT (INCLUDING NEGLIGENCE) FOR ANY DAMAGES RESULTING FORM USE OF A PRODUCT OR SERVICE DESCRIBED HEREIN, OR FOR ANY INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THE ABILITY OR INABILITY TO USE THE PRODUCTS, TO THE FULL EXTENT THESE DAMAGES MAY BE DISCLAIMED BY LAW.

Some states and other jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, or limitation on the length of an implied warranty, so the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights, which vary from jurisdiction to jurisdiction.

Motorola products or services are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product or service could create a situation where personal injury or death may occur.

Should the buyer purchase or use Motorola products or services for any such unintended or unauthorized application, buyer shall release, indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the designing or manufacture of the product or service.

Contact Information

Motorola, Inc.
IDEN Subscriber Group
Phone: 1-800-453-0920
URL: https://commerce.motorola.com/idenonline/ideveloper/index.cfm

Acronyms and Definitions

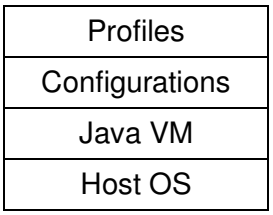
Acronym	Terminology	Definition
AMS	Application Management Software	A subsystem that manages the life cycle of MIDlets.
API	Application Programming Interface	A set of classes, interfaces, and methods that can be used for creating an application.
CSD	Customer Specific Data	Contains frequency list for the carriers.
-	DateBook	The native platform's Date Book.
-	DateBookEvent	An event in the native Date Book.
GPS	Global Positioning System	Identifies the location of a device.
-	Java Volume	The volume tables for Java-produced audio. The tables are for Java Earpiece and Java Speaker. When Java has focus, pressing the volume keys affect the Java Volume.
-	Java Audio Path, Java Audio Route	Similar to the two-way radio feature, Java has a setting that routes audio through the earpiece or the loud speaker.
LWT	Lightweight Windowing Toolkit	A Motorola-proprietary GUI framework.
MIDP 2.0	Mobile Information Device Profile (MIDP) v2.0	Sun's J2ME™ profile for connected, mobile devices, such as this phone.
NMEA	National Marine Electronics Association	
-	PhoneBook	An entry in the platform's PhoneBook.
-	PhoneBookBookEvent	Interchangeably used as DateBook data.
PKI	Public Key Infrastructure	
RMS	Record Management System	One of the mechanisms for persistent storage of data on this phone.
SSL	Secure Socket Layer	
UDM	User Data Manager	Class for accessing the native databases with user information.
UFMI	Universal Fleet Member Identifier	Identifier used in private dispatch calls.
-	Unauthorized MIDlet	A MIDlet that does not have a valid certificate to access Motorola OEM domain packages.
WMA	Wireless Messaging API	

1. Introduction

Motorola's iDEN i325 multi-communication device includes the Java 2 Platform, Micro Edition, also known as the J2ME™ platform. The J2ME™ platform enables developers to easily create a variety of applications ranging from business applications to games. Prior to its inclusion, services and applications that reside on small consumer devices like cell phones could not be upgraded or added without significant effort. By implementing the J2ME™ platform on devices like the i325 phone, service providers as well as customers can easily add and remove applications, allowing for quick and easy personalization of each device. This section of the guide provides a quick overview of the J2ME™ environment and the tools that can used to develop applications for the Motorola i325 phone.

1.1. The Java 2 Platform, Micro Edition (J2ME™ Platform)

The J2ME™ platform is a highly optimized, extended subset of the Java 2 Platform, Standard Edition, designed for deployment on small devices such as cell phones and pagers. It includes both a set of APIs and a virtual machine and is designed in a modular fashion allowing for scalability among a wide range of devices. The J2ME™ architecture contains three layers consisting of the Java Virtual Machine, a Configuration Layer, and a Profile Layer. The Virtual Machine (VM) supports the Configuration Layer by providing an interface to the host operating system. Above the VM is the Configuration Layer, which can be thought of as the lowest common denominator of the Java Platform available across devices of the same "horizontal market." Built upon this Configuration Layer is the Profile Layer, typically encompassing the presentation layer of the Java Platform.



The J2ME™ Platform Architecture

The Configuration Layer used in the i325 phone is the Connected Limited Device Configuration 1.0 (CLDC 1.0) and the Profile Layer used is the Mobile Information Device Profile 2.0 (MIDP 2.0). Together, CLDC and MIDP provide common APIs for I/O, simple math functionality, UI, and more.

For more information on J2ME™ and J2SE™, see http://java.sun.com/.

1.2. The iDEN J2ME™ Platform

Functionality not covered by the CLDC and MIDP APIs is left for individual OEMs to implement and support. By adding to the standard APIs, manufacturers can allow developers to access and take advantage of the unique functionality of their devices.

The i325 phone contains OEM APIs for a wide variety of extended functionality ranging from enhanced UI to advanced data security. While the i325 phone can run any application written in standard MIDP 2.0 or MIDP 1.0, it can also run applications that take advantage of the unique functionality provided by these APIs.

1.3. Resources Available on the i325 Phone

The i325 phone allows access to a richer set of resources than our previous iDEN Java capable phones. The changes range from a larger heap for Java applications to the presence of a 12-bit TFT color display. All of the enhancements allow for more compelling and advanced Java applications to be created. In addition to increasing resources present on the device, new APIs to access other

device resources were added. These new APIs allow a Java application to leverage other capabilities of the device that are currently not accessible through standard MIDP 2.0 and CLDC APIs.

Table 1. Resources In The i325 Phone

Display	
Resolution	96 x 65
Color Depth	12-bit color
Networking	
Max TCP Sockets*	14
Max HTTP connections*	4
Max UDP sockets*	21
File & RMS	
Open Files/RMS per MIDlet**	5
Floating Open Files/RMS for MIDlet***	16
Java VM	
Heap Size	1150 KB
Program Space	1152 KB
Max Resource Space*	1280 KB
Recommended Maximum JAR Size	500 KB
Multimedia	
Number of concurrent MIDI files playing	1

*: These resources are shared with the rest of the phone. Actual resources available to MIDlets can vary.

**: i325 phone guarantees that each MIDlet can at least open 5 files/RMS.

***: At first come fist serve bases.

1.4. New Features on the i325

1.4.1. Concurrency

The i325 supports the concurrent execution of up to three MIDlets at a time. The three MIDlets must be from different MIDlet Suites. You can't concurrently execute two MIDlets from the same MIDlet Suite.

Only one of the MIDlets can be in the foreground at once, leaving the other two suspended in the background. As with other iDEN products, the MIDlets in the background can still execute while they're suspended. Now more than ever, it is recommended that MIDlets be written in such a way that they release any resources such as files, large temporary heap storage, and threads.

The i325 uses a single instance of the VM to run all three MIDlets. This means that all of the MIDlets' threads are scheduled together in a round robin fashion, but the time slices vary. The threads belonging to the foreground (active) MIDlet have the largest time slices. The threads belonging to the background (suspended) MIDlets have smaller time slices. The thread's actual time slice is calculated based on this foreground/background designation as well as the thread's priority. A MIDlet can set that thread priority by using the `Thread.setPriority()` method as described in the CLDC specification. This allows the MIDlet to customize the performance of it's own threads in respect to the other threads with the same foreground/background designation, basically meaning that no threads in a background MIDlet can be a higher priority than the threads in the foreground MIDlet.

All three MIDlets also operate out of the same heap. This raises the possibility of an out of memory exception because there are other applications whose memory usage is not known. The best way to safeguard against this is for application developers to minimize the amount of heap they consume while their MIDlet is in the background. Another way to avoid an out-of-memory problem is to catch the OutOfMemoryError while creating large resource objects. This prevents unexpected behaviors from the MIDlet.

Although MIDlets can run concurrently, they still don't have an awareness of each other. The purpose of the concurrency feature is not cooperation amongst MIDlets. The purpose is to allow the user to run multiple MIDlets, such as a game in the foreground and an instant messenger application in the background.

1.4.2. Multiple Key Presses

The i325 phone supports the ability to have multiple key presses passed to a Java application. This allows for applications to receive notification of another key being pressed while one key is still being pressed. Due to hardware limitations, only 2 keys presses can be guaranteed at any given time, while many key combinations of more keys can be pressed at the same time.

Multiple key press support works as follows on the i325:

When a key is pressed, the application's `keyPressed()` method is called

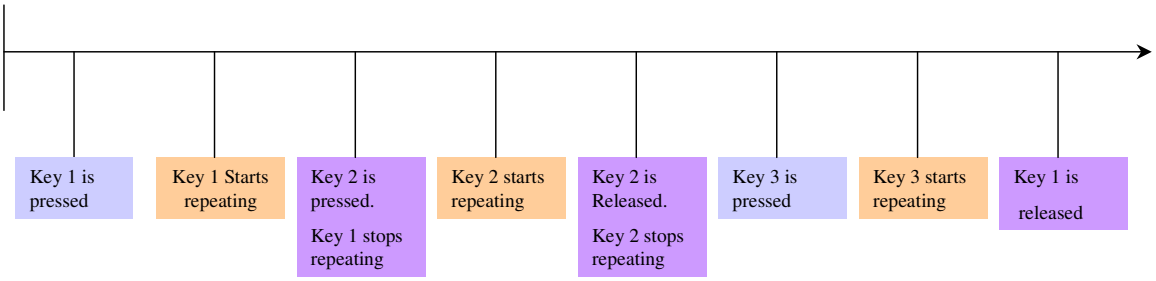
After 650ms, the application's `keyRepeated()` method is called every 250 ms

When a second key is pressed, it stops the first key from repeating. The second key will generate another call to the application's `keyPressed()` method.

After 650ms, the application's `keyRepeated()` method is called every 250 ms with the second key's key code

When either of the keys are released, the application's `keyReleased()` method is called with the key code of the key that was released

If two keys are currently pressed and if the key that is released is the repeating key, no more calls to the application's `keyRepeated()` method occur until another key is pressed.



Key press timeline

1.4.3. Timezone and Daylight Savings Support

On previous phone models no local timezone or daylight savings time information was available. However, the i325 provides the default timezone as the local timezone, and daylight savings information is also available through the CLDC TimeZone class. For instance, if it is 5:00PM locally, local timezone is GMT-05:00, then `System.currentTimeMillis()` returns 10:00PM (9:00PM if daylight savings is in effect).

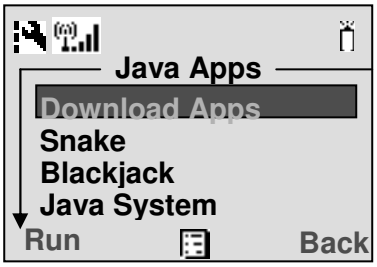
In addition, `System.currentTimeMillis()` now returns accurate GMT time. On previous products, default timezone is always GMT, and `System.currentTimeMillis()` returns local time. For instance, on previous products: If it is 5:00PM locally, then `System.currentTimeMillis()` returns the millisecond value of 5:00PM (no daylight savings information available).

2. Application Management

2.1. MIDlet Lifecycle

A MIDlet's lifecycle begins once its MIDlet suite is downloaded to the device. From that point, the Application Management Software (AMS) manages the MIDlet suite and its MIDlets. The user's primary user interface for the AMS is the Java Apps feature built into the device's firmware.

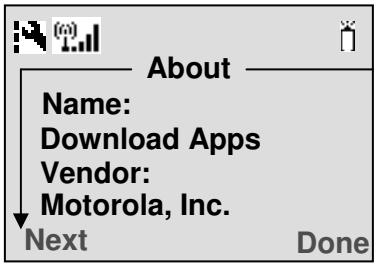
From the Java Apps feature, the user can see each MIDlet suite on the device or access the Java System menu item. If a MIDlet suite has only a single MIDlet, then the MIDlet's name is displayed in the Java Apps menu for that MIDlet suite. Otherwise the MIDlet suite name is displayed. Then when that MIDlet suite is highlighted, the user can open the MIDlet suite and view the MIDlets in that MIDlet suite.



The Java Apps Menu

From the Java Apps menu, the user can highlight a MIDlet suite and bring up the About dialog for that MIDlet suite. The About dialog contains:

- MIDlet Suite Name
- MIDlet Suite Vendor
- MIDlet Suite Version
- JAR Size (not installed only)
- The number of MIDlets in the MIDlet Suite
- MIDP and CLDC version requirements (not installed only)
- Flash usage, Program and Data Space (installed only)



About Properties for a MIDlet

2.2. MIDlet Suite Installation

From the Java Apps menu, the user can install MIDlet suites. A MIDlet suite must be installed before any of its MIDlets can be executed. Installation involves extracting the classes from the JAR file and creating an image that will be placed into Program Space. The resources are then extracted from the JAR file and placed into Data Space. The JAR file is then removed from the device, thus freeing up some Data Space where it was originally downloaded.

The space savings from removing the JAR file is one advantage of Installation. However, perhaps an even greater advantage is that class loading is not done during run time. This means that a MIDlet won't experience slow-down when a new class is accessed. Furthermore, the MIDlet won't have to share the heap with classes have been class-loaded from the JAR file.

2.3. MIDlet Suite De-installation

An installed MIDlet can be removed from the device only by de-installing it from the Java Apps menu. De-installing a MIDlet suite removes the installed image from Program Space. The resources are then removed from Data Space along with the JAD file.

2.4. MIDlet Suite Updating

When a MIDlet suite is de-installed, all of its resources are removed including any resources that were created by the MIDlets in the suite, such as RMS databases. If a user gets a new version of a MIDlet suite, the user can simply download that new version to the device that has the older version. Once that new version is downloaded, the user has the option to update the MIDlet suite. This de-installs the old version and immediately installs the new MIDlet suite. The only difference is that the device asks the user whether resources such as RMS databases should be preserved while de-installing the old version. This prompt occurs only if such resources exist.

Such a scheme places the burden of compatibility on the developer. A newer version of the MIDlet suite should know how to use, upgrade, or remove the date in the RMS databases that the older versions created. This idea of forward compatibility should also extend to backward compatibility, because the device allows a user to replace a version of a MIDlet suite with an older version of that MIDlet suite.

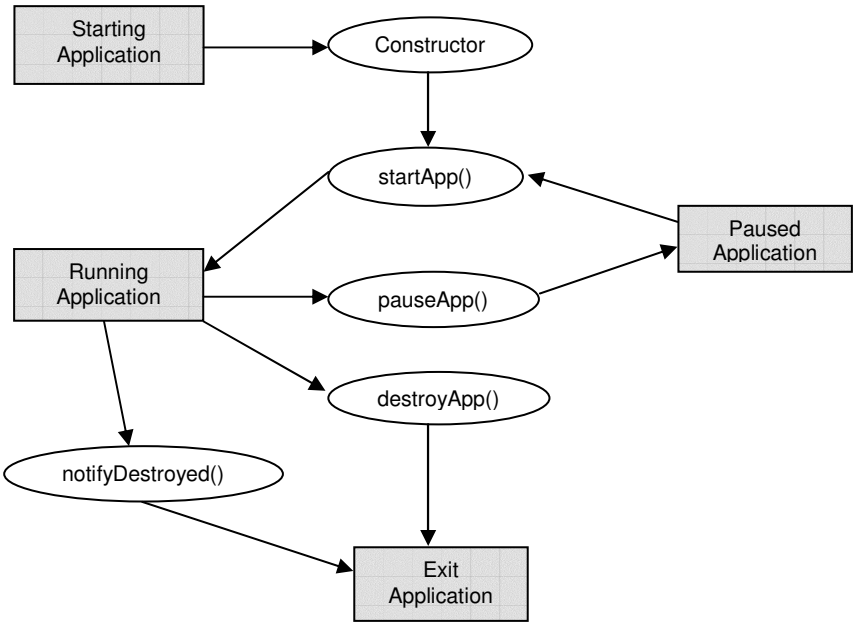
2.5. Starting, Pausing, and Exiting

2.5.1. AMS Control of MIDlet State Transitions

A MIDlet has three different states: destroyed, active, and paused. A MIDlet's natural state is destroyed. The AMS typically controls the transition through these states. When a user decides to launch a MIDlet, the device puts up a screen indicating that the MIDlet is transitioning through these states. The AMS controls the MIDlets through those states by calling the MIDlet's methods, `startApp()`, `pauseApp()`, and `destroyApp()`.

MIDlet Starting Screen

First, the constructor of the MIDlet's class that extends MIDlet is invoked. Then its `startApp()` method is called to indicate that it's being started. The MIDlet has focus when its `startApp()` method finishes execution. If a MIDlet takes too long initializing state variables and preparing to be run in its constructor or `startApp()` methods, it may appear to be stalled to users.



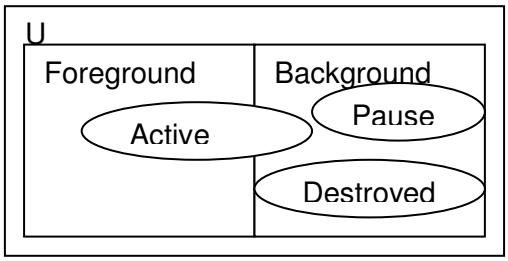
MIDlet State Transitions

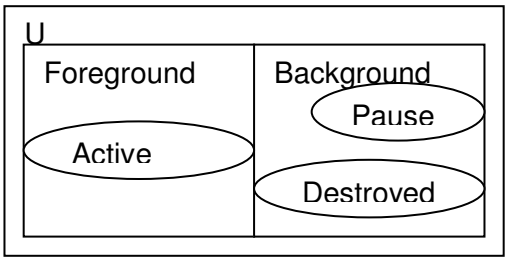
Table 2. State Transition Methods

Method	Caller	Purpose
Constructor	AMS	Initializes the MIDlet – should return quickly
`startApp()`	AMS	1. The `startApp()` method is called to start the application either from a newly constructed state or from a paused state. 2. If the `startApp()` is called from a paused state, the MIDlet should not re-initialize the instance variables (unless that's the desired behavior). 3. The `startApp()` method may be called multiple times during the lifespan of the MIDlet. 4. The MIDlet may set the current display to its own Displayable from the `startApp()` method, but is shown only after the `startApp()` returns. 5. When exiting a suspended application, the KVM calls `startApp()` first followed by a call to `destroyApp()`
`pauseApp()`	AMS, MIDlet	1. The `pauseApp()` method is called from either AMS or from within the MIDlet. 2. The `pauseApp()` should pause active threads, and prepare for `startApp()` to be called. 3. If the application is to be resumed with a screen other than the present, then the Displayable should be set current in the `pauseApp()`.

`destroyApp()`	AMS	1. The `destroyApp()` method is called from AMS and signals the MIDlet to clean up any resources and prepare for termination. For example, open RMS records should be closed, threads should be stopped, and any other housekeeping chores should be performed. 2. The MIDlet should not call `destroyApp()`.
`notifyDestroyed()`	MIDlet	1. The `notifyDestroyed()` method is called by the MIDlet to exit and terminate itself. 2. All housekeeping such as stopping active threads and closing RMS records should be performed before calling `notifyDestroyed()`. 3. `notifyDestroyed()` notifies AMS to terminate the calling MIDlet.

Focus is an important concept. On a device without a windowing system, only one application can have focus at a time. When an application has focus, it receives keypad input, and has access to the display, speakers, LED lights, vibrator, and so on. MIDlets share focus with the system user interface. That user interface is a higher priority than the MIDlet, so the MIDlet will immediately lose focus when the system needs to handle a phone call or some other interrupt.



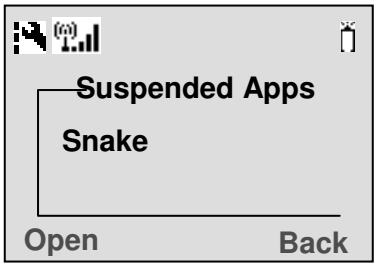


Generic MIDP vs. iDEN Devices

On iDEN devices, the concept of focus correlates directly with the MIDlet state. For example, when a MIDlet loses focus because of a phone call, the MIDlet is immediately suspended. Conversely to the example of starting the MIDlet, the MIDlet loses focus immediately, then its `pauseApp()` method is called. Standard MIDP allows multiple MIDlets, where a MIDlet can be active in the foreground or active in the background. However, on the i325 phone, an active MIDlet implies foreground and a paused MIDlet implies background.

The paused state is not clearly defined by MIDP. The only requirement placed on the device manufacturer is that a paused MIDlet must be able to respond to network events and timer events. On iDEN devices, the paused state simply implies that the MIDlet is in the background as mentioned above, but it doesn't force any of the threads to stop execution. Essentially, a paused MIDlet is a MIDlet without focus and whose `pauseApp()` method has been called. It's up to the developer to control their threads, such as making them sleep for longer periods, completely pausing game threads, or terminating threads that can be restarted when the MIDlet is made active again.

Similarly to the example of losing focus immediately before the `pauseApp()` method is called, a MIDlet's focus is also immediately lost immediately before its `destroyApp()` method is called. It's interesting to note how an iDEN device manages the transition to the destroyed state. The user's opportunity to exit a MIDlet using the AMS, is from the MIDlet's Suspended screen. Typically a MIDlet is suspended, then the user exits it. Even though it appears the MIDlet is going immediately from the paused state to the destroyed state, it actually transitions through the active state first, but it never gains focus during that transition.



MIDlet Suspended Screen

2.5.2. MIDlet Control of MIDlet State Transitions

A MIDlet has a lot of flexibility to control its own state. A MIDlet can call its own `startApp()`, `pauseApp()`, and `destroyApp()` methods. However those are the methods that the AMS uses to indicate a state transition to the MIDlet, so this won't actually cause the state transition. The MIDlet can simply call those methods if it wishes to perform the work that it would typically do during that state transition.

There is another set of methods that the MIDlet can use to cause the actual state transitions. They are `resumeRequest()`, `notifyPaused()`, and `notifyDestroyed()`. Since the system user interface has priority, a MIDlet cannot force itself into the active state, but it can request that it be resumed with `resumeRequest()`. If the system is not busy, then it will automatically grant the request. However if the device isn't in the idle screen, then it displays an alert dialog to ask whether the user would like to resume the MIDlet. If the user denies the request, the MIDlet is not notified. If the user grants the request, the MIDlet's `startApp()` method is called, and it gains focus when that method finishes.

The MIDlet does have more control when it decides that it wants to be paused or destroyed. It would perform the necessary work by calling its own `pauseApp()` or `destroyApp()` method, then it notifies the AMS of its intentions by calling `notifyPaused()` and `notifyDestroyed()` appropriately. Once notified, the AMS changes the MIDlet's state and revokes focus.

2.6. Java System

Besides managing MIDlet suites from the Java Menu, you can also perform system maintenance. The Java System feature gives statistics about the system such as:

CLDC Version

MIDP Version

Data Space Free

Program Space Free

Total Heap Space

Besides getting statistics, you can reset the Java System or format the Java System. Resetting the Java System simply re-initializes the components of each MIDlet suite as if the device was just powered up. Formatting the Java System actually removes every MIDlet suite by completely formatting the components of each MIDlet suite. These features can be accessed on the i325 by pressing the Menu key while highlighting Java System. Note that "Format System" has been renamed to "Delete All" on the i325.

2.7. Java From Main Menu

Previously the Java Apps menu was the only interface into the Java functionality of an iDEN device. However, starting with the i95cl, the main menu has been enhanced to allow the user to add links to MIDlets or entire MIDlet suites to the main menu. When a MIDlet is added to the main menu, the name of the individual MIDlet is used for the main menu text. When a MIDlet suite has multiple MIDlets, the MIDlet suite itself can be added and the main menu text will be the MIDlet suite name. When a MIDlet suite is selected from main menu, the device opens that MIDlet suite and displays the MIDlets as if it were opened from the Java Apps menu.

2.8. Personalizing the Native UI

The i95cl let the user personalize their native user interface by adding MIDlets and MIDlet suites to the main menu. The i325 takes it a step further by allowing the user integrate their MIDlets and MIDlet suites in a number of ways.

Home Screen Soft keys

Home Screen Navigation Keys

Shortcuts

Power Up App

Datebook Events

The first three items are additional places where the user can press a key or key sequence to launch the MIDlet or MIDlet suite. The Power Up App feature allows you to launch a MIDlet suite when the phone powers up. The user can also specify a MIDlet to be launched when a Datebook event occurs.

These various features are options for the user, but as a developer you may want to encourage users to set your MIDlet up with one of these features. However, from your MIDlet you can specify a MIDlet to be added to a datebook event through the Datebook API (see “DateBook” on page 81).

2.9. The miniJIT

The miniJIT is iDEN's Just In Time (JIT) compiler technology that optimizes computation intensive code within a MIDlet suite. The miniJIT works by identifying code that can be optimized and compiling the Java bytecodes to native code because native code executes faster. During the compilation, the miniJIT can also eliminate some validity checks required in the Java interpreter. Finally, the miniJIT can accelerate method calls, one of the most common constructs in any Java application, using a technique called the fast method call.

The miniJIT is made of two separate components:

The ahead of time code analyzer

The optimized Java to native compiler

The ahead of time code analyzer identifies the performance crucial code in a MIDlet suite during the installation of the MIDlet suite. The optimized compiler compiles the code that the code analyzer identifies as performance crucial into native code. During the installation of the MIDlet suite, the compiler also searches for methods that can use the miniJIT's fast method call capabilities and modifies the code appropriately to use this capability.

Unlike most JIT compilers, the miniJIT compiles code only at install time. Most JIT compilers compile the code when the application is running. The miniJIT does not do so because it runs on a deeply embedded device. By compiling ahead of time, the miniJIT also eliminates the unpredictability of the dynamic behavior of most JIT compilers.

As with most optimization technologies, there are specific trade-offs in using the miniJIT. The miniJIT increases the size of the installed MIDlet suite by approximately 20% on device since it compiles the

compact Java bytecodes to native code. Using the miniJIT also increases install time of the MIDlet suite since the code analyzer and compiler execute during this time.

While the miniJIT does optimize computationally intensive tasks, many common capabilities are already optimized for the iDEN platform and therefore the miniJIT provides little or no additional benefit. These capabilities include:

Graphics (LCDUI)

Image Processing

RMS

2.9.1. Using the miniJIT

By default, the miniJIT does not compile code during installation. To use the miniJIT, the following line must be added to the JAD file of the MIDlet suite:

```
iDEN-MIDlet-miniJIT: on
```

The `iDEN-MIDlet-miniJIT` attribute is checked during the installation of a MIDlet suite to determine if the miniJIT should be used. If the value is set to "on", the miniJIT is used to compile the MIDlet suite code. If the attribute is absent or set to "off", the miniJIT is not used.

2.9.2. Tips

The miniJIT optimizes only the code within the MIDlet suite.

The miniJIT excels at optimizing loops and computation intensive code.

The miniJIT can use the fast method call if the method that is being called meets the following conditions:

- The method must be performance crucial.
- The method is not abstract or synchronized.
- The method does not have any exception handlers.
- The method must be static or have no overriding methods.
- The method must not allocate any memory from the Java heap.
- All the methods that this method calls must also meet all the criteria listed here.

The miniJIT will not improve the performance of LCDUI code, loading of Images, and RMS code. It may improve the performance of the supporting code within the MIDlet suite.

In order to use the miniJIT, the MIDlet suite must be installed.

As with all optimization techniques, only on-device testing lets a developer predict the changes to the user experience the optimization may have.

3. Developing, Packaging, and Deploying J2ME™ Applications

3.1. Developing – Tools and Emulation Environments

In order to develop applications for a J2ME™ enabled device, a developer needs some specialized tools to improve development time and prepare the application for distribution. There are several tools available in the space, so this overview is included to help enable developers to make an informed decision on these tools.

3.1.1. Features to Look For

Numerous tools for developing J2ME™ applications are readily and freely available on the market. Some of their features include:

Class libraries. J2ME™ tools include class files for the standard CLDC/MIDP specifications and may also contain class files needed to compile device specific code.

> One of the main characteristics of the MIDP 2.0 standard is the lack of device specific functionality. As a solution, many MIDP 2.0 device manufacturers have implemented Licensee Open Classes that provide the features requested by developers. In order to take advantage of these APIs, choose an SDK that natively supports them or one that can be upgraded to support them.

API documentation. In addition to providing the class files, most SDKs include reference documentation for the supported APIs. These documents, typically found in either a HTML or PDF format, cover the standard CLDC/MIDP specifications as well as the device specific APIs.

Emulation environment. Although not an absolute necessity if the device is available, most toolkits provide this functionality for multiple devices. The main benefits of an emulation environment are the reduction in development time as well as the ability to develop for devices not yet on the market. The extent to which the toolkits emulate the device can vary greatly.

> If most of the development is going to take place on the device, then this may not be a big consideration, but if access to the target device is limited or unavailable, accurate emulation is a must. Look for accuracy in the font representation, display dimensions, and pixel aspect ratio, as many wireless devices do not have square pixels.
>
> Along the same lines as accurate look and feel, the tool should also provide accurate performance emulation. A comprehensive tool should provide individual adjustments for performance aspects such as network throughput, network latency, persistent file system access time, and graphics performance. Ideally, these attributes should not only match the target device, but also have the ability to be manually adjusted.

Application packaging utility. Most SDKs automatically package the application for deployment onto the target device. Although many tools include this feature, flexibility varies widely. Look for a tool that generates both the manifest and JAD files with the required tags as well as custom tags. The packaging steps required to deploy an application on the i325 are described in a subsequent section.

3.2. Packaging – Putting the Pieces Together

Once an application has been tested on an emulator and is ready for testing on the actual device, the next step is to package the application and associated components into a JAD/JAR file pair. The files contain both the MIDlet's executable byte code along with the required resources. Although this process is automatically performed by most SDKs and IDEs supporting J2ME™, the steps are explained and outlined here.

3.2.1. Compiling .java Files to .class Files

Compiling a J2ME™ application is no different the any other J2SE™/J2EE™ application. By adding the CLDC/MIDP files (whether functional or stubbed out) to the classpath, any standard Java compiler that is JDK1.2 compliant or greater is sufficient to produce .class files suitable for the preverification step.

3.2.2. Preverifying .class Files

Class files destined for the KVM must undergo a modified verification step before deployment to the actual device. In the standard JVM found in J2SE™, the class verifier is responsible for rejecting invalid classes, classes that are not compatible, and classes that have been modified manually. Since this verification step is processor and time intensive, it is not ideal to perform verification on the device. In order to preserve the low-level security model offered by the standard JVM, the bulk of the verification step is performed on a desktop/workstation before loading the class files onto the device. This step is known as preverification.

During the preverification step, the class file is analyzed and a stack map is appended to the front of the file. Although this may increase the class file size by approximately 5%, it is necessary to ensure the class file is still valid when it reaches the target device. The standard J2SE™ class verifier ignores these attributes, so they are still valid J2SE™ classes.

3.2.3. Creating a Manifest File with J2ME™ Specific Attributes

In addition to the class files, a manifest file for a MIDlet needs to be created. Although most J2ME™ tools will auto generate the manifest file, it can also be created manually using a plain text editor. The following is a sample manifest file for a HelloWorld MIDlet:

```
MIDlet-Name: HelloWorld
MIDlet-Version: 1.0.0
MIDlet-Vendor: Motorola, Inc.
MIDlet-1: HelloWorld, , com.motorola.midlets.helloworld.HelloWorld
MicroEdition-Profile: MIDP-2.0
MicroEdition-Configuration: CLDC-1.0
```

The device's AMS uses the manifest file to determine the number of MIDlets present within the suite as well as the entry point to each MIDlet. Additionally, the manifest files may contain optional tags that are accessible by the MIDlets within the MIDlet suite. For more information, refer to the MIDP 2.0 specifications.

Keep in mind these notes when creating a manifest file**:**

The following attributes are mandatory and must be duplicated in both the JAR file manifest and the JAD file. If the attributes are not identical, the application will not install.

```
MIDlet-Name
MIDlet-Version
MIDlet-Vendor
```

The manifest contains `MIDlet-<n>` arbitrary attributes each describing a MIDlet in an application suite.

The MIDlet-1 attribute contains three comma-separated fields: the application name, the application icon, and the application class file (entry point). The name is displayed in the AMS user interface to represent the nth application.. The application class file is the class extending the `javax.microedition.midlet.MIDlet` class for the nth MIDlet in the suite.

The manifest file is case sensitive.

The manifest must be saved in a file called `MANIFEST.MF` (case sensitive) within the `meta-inf` directory.

3.2.4. JARing .class Files and Other Resources

Once the application is ready to be packaged for the device, its class files and associated resources must be bundled in a Java Archive (JAR) file. The JAR file format enables a developer to bundle multiple class files and auxiliary resources into a single compressed file format. The JAR file format provides the following benefits to the developer and end-user:

Portability. The file format is platform independent.

Package Sealing. All classes in a package must be found in the same JAR file.

Compression. Files in the JAR may be compressed, reducing the amount of storage space required. Additionally, the download time of an application or application suite is reduced.

3.2.5. Creating the JAD File

Although the Java Application Descriptor (also known as an Application Descriptor File) is optional in the MIDP 1.0 specification, J2ME™ applications targeted for Motorola iDEN devices must include a JAD/JAR pair. The following is a sample JAD file for a simple HelloWorld application.

```
MIDlet-Name: HelloWorld
MIDlet-Version: 1.0.0
MIDlet-Vendor: Motorola, Inc.
MIDlet-Jar-URL: http://www.motorola.com
MIDlet-Jar-Size: 1939
MIDlet-Description: A sample HelloWorld application.
```

The JAD file may be created with any text editor and saved with the same file name prefix as the JAR file. The mandatory `MIDlet-Name, MIDlet-Version, MIDlet-Vendor` must be duplicated from the JAR file manifest. JAR files containing manifests that do not match the JAD file will not be installed.

Keep these notes in mind when creating the JAD file:

The file names of the JAD and JAR are required to be identical except for the file extension. For example the JAR file for the `HelloWorld.jad` must be named `HelloWorld.jar`.

The JAD file is case sensitive. All required attributes in the JAD file must start with "`MIDlet-`" followed by the attribute name.

The total file length is limited to 16 characters, including the `.jad` and `.jar` extensions. For example, `HelloWorld.jad` occupies 14 characters.

The `MIDlet-Jar-Size` must contain the accurate size of the associated JAR file. The number is in bytes.

It's also important to note that these fields must have associated values with them. Example: "`MIDlet-Name: `" is not valid but "`MIDlet-Name: Snake`" is valid.

For more information regarding the JAD file, please refer to the MIDP 1.0 specification.

3.3. Desktop to Device

Now that the application is packaged, it is ready to be loaded on to the device. Applications are categorized into two distinct categories: networked and walled garden. Applications that fall into the networked category use services such as packet data (examples include HTTP, sockets, UDP, etc) to retrieve information from a remote server. Typical examples of this include applications like web browsers that use packet data services to retrieve content from remote web servers located on the open Internet. Walled garden applications, on the other hand, are stand-alone applications that do not

make use of packet data services. These applications contain all the necessary data locally. Typical applications in this category include games, conversion utilities, etc.

Motorola distributes two Java Application Loading tools: JAL Lite and WebJAL. Walled Garden applications can be loaded with either WebJAL or JAL Lite. Networked applications, on the other hand, can only be loaded via the WebJAL utility. Both tools are available at www.idendev.com.

3.4. Debugging – Terminal Interface

As with most application development lifecycles, 10 percent of the time is spent doing the first 90 percent and 90 percent of the time is spent doing the last 10 percent. Since debugging the application is inevitable, setting up a debug environment on the phone is quite desirable.

3.4.1. HyperTerminal

The tool typically used to debug on the device is HyperTerminal, found at http://www.hilgraeve.com. HyperTerminal is also included with Windows NT and is accessible under the accessories menu. The HyperTerminal acts as a terminal for J2ME™ applications residing on the device. For example, in the standard Java 2 Platform, if a

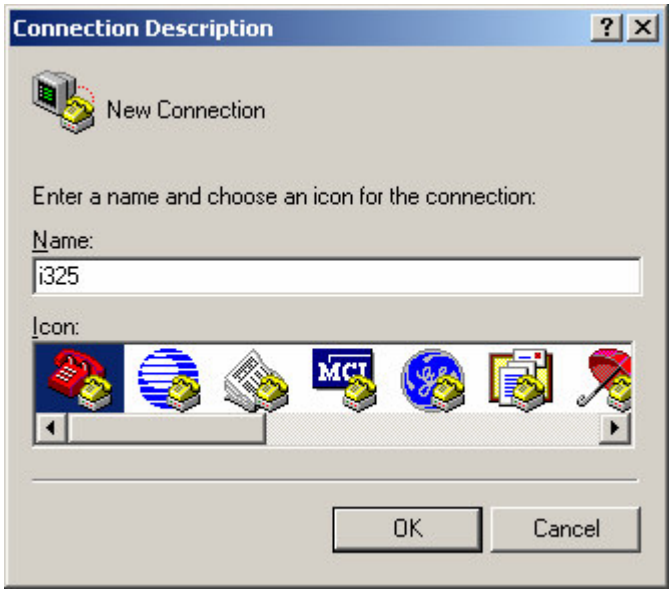
```
System.out.println("Something")
```

is issued, the output is displayed in the terminal from which the Java application was launched. The same reasoning applies to applications residing on the i325. The following instructions describe the necessary steps required to setup HyperTerminal for the i325. Note – a data cable is also required for debugging.

Start the HyperTerminal applications by selecting Start -> Accessories -> HyperTerminal -> HyperTerminal from the "Start" menu.

From within the HyperTerminal application, select the File -> New Connection menu item from the drop down menus located at the top of the application.

Choose a name as well as an icon for the new connection, something like "i325", and click on the "Ok" button.

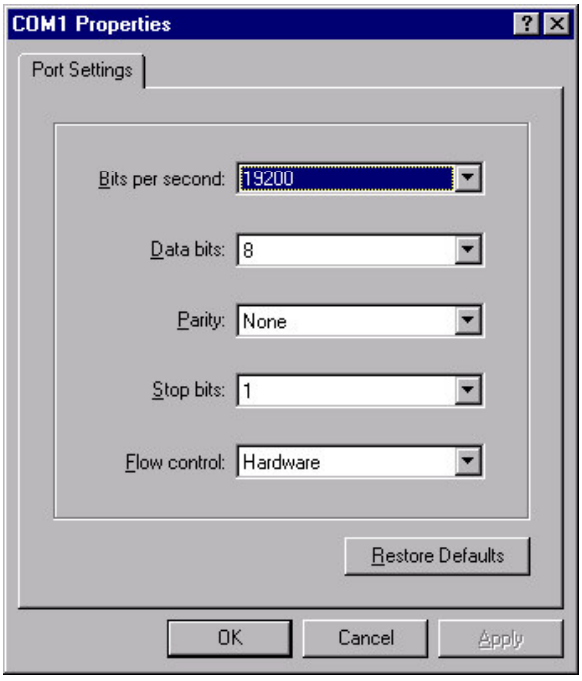


Creating a New HyperTerminal Connection

Select the Communication port the data cable is connected to, typically COM1 or COM2.

Setting The Connection Communication Port

A properties dialog box appears. Configure the bits per second to coincide with the Baud Rate set on the i325. Set the Data bits to 8, Parity to None, Stop bits to 1, and Flow control to Hardware.



Setting The Communication Port Properties

Once all the parameters have been set, save the profile. To save the profile, select File->Save. The profile is saved as the connection name plus an .ht extension. For the example, the profile is saved as i325.ht.

The profile can be launched from the Start->Accessories>HyperTerminal->i325 menu choice.

3.4.2. Java Debug

To turn on Java debug, the following AT commands must to be issued to the i325 phone via the HyperTerminal in one of two ways:

3.4.2.1. Keyboard Input

From the keyboard, type the following AT command to turn on Java debug.

```
AT+WS46=252;+WS45=0;+IAPPL=2;D
```

The previous command turns on the Java debug statements for the current HyperTerminal session.

3.4.2.2. Text File Transfer

The previous AT command listed above can also be saved in a text file and transferred to the i325 via the HyperTerminal. To transfer the text file, select the Transfer->Send Text File menu command or press <alt> + t.

NOTES: Java debug is turned on only for a particular HyperTerminal session. If the data cable is disconnected or 'Disconnect' button on the HyperTerminal is pressed, the previous sequence must be repeated to re-enable Java debug.

Debug information may not appear on the HyperTerminal if extra control characters are buffered. Type "AT" in the HyperTerminal to check the connection status. If an "OK" is returned then the connection does not contain buffered characters. To turn the echo on, type "ATE1".

Ensure the data communication rate for the i325 phone coincides with the bits per second on the HyperTerminal. If the data rates are different, debug messages will not appear

3.4.3. Method Tracing

Once Java Debug is turned on, method tracing can be turned on. To see the menu of commands available, type <m> on the keyboard. The following is a sample of navigating through this menu.

```
At
OK
AT+WS46=252;+WS45=0;+IAPPL=2;D
OK
m
   M           - Menu
  TM [On/Off]  - Trace Methods
 TMM [On/Off]  - Trace Motorola Methods
 TMJ [On/Off]  - Trace J2ME™  Methods

>tm on
Method Tracing On
>tmm on
Motorola Method Tracing On
>tmj on
JAVAX Method Tracing On
>
```

Motorola method tracing tracks any methods within Motorola extensions to the base classes. J2ME™ method tracing will track all method calls within the standard J2ME™ methods

3.4.4. Debug Statements

Debugging J2ME™ applications is very similar to debugging typical Java 2 Platform applications. The most common method of debugging applications is to place `System.out.println()` statements in strategic locations. A simple way to create a debug and production version of the application at compile time is to encapsulate the `System.out.println()` statements in `if…then` clauses, with the `if` conditional checking a static Boolean variable. See the following code example:

```
class TestClass{

      private static final boolean debug = true;

      public static void main(String args[]){
            if(TestClass.debug){
                  System.out.println("Debug turned on");
                  }
            }
}
```

By changing the debug flag at compile time, debug statements can be easily turned on and off. For more information on debugging J2ME™ and Java applications, see

http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/collect.html

3.5. Beyond Standards

In addition to supporting the MIDP 2.0 specification, the iDEN Java platform contains extensions in the JAD file that reflect the increased capabilities of the device. These new extensions were created to support features such as Internationalization (I18n). Although these specifications are beyond the standard MIDP 2.0, their existence is necessary to provide features requested by both the international and domestic development community. The following sections detail the specifics of each extensions as well as format and syntax.

3.5.1. Making it Global – Internationalization (I18n)

This phone is an I18n-ized J2ME™ platform, enabling developers to provide and display different languages. Since the MIDP 2.0 specification does not address the issue of multi-language support for MIDlet attributes, iDEN specific attribute tags were created to provide developers with this functionality. Prior to the I18n-ized J2ME™ platform, developers could only display MIDlet suite, vendor, and friendly MIDlet names in one language, and only in basic and extended ASCII characters.

Since, the MIDlet suite, vendor, and friendly names can be in different languages, the ADF file must support multi-language friendly format such as Unicode. The UTF-8 format is used to support multiple languages. Using this I18n-ized J2ME™ platform, you can present English, Spanish, French, Portuguese, and Korean text for your MIDlet attributes. Please note that the domestic phone will only support English, French, Portuguese, and Spanish.

For more information on UTF-8 format, visit: www.unicode.org

The following are additional MIDlet attributes in the I18n-ized J2ME™ platform:

Table 3. I18n MIDlet Attributes For The i325 Phone

Attribute Name	Attribute Description
`iDEN-MIDlet-Name-xx`	The name of the MIDlet suite that identifies the MIDlets to the user in xx language.
`iDEN-Vendor-xx`	The organization that provides the MIDlet suite in xx language.
`iDEN-MIDlet-xx-<n>`	The name, icon and class of the nth MIDlet in the JAR file separated by a comma in xx language.
xx represents the language code. For example, en for English, es for Spanish, pt for Portuguese, and fr for French	

The following ADF for I18n-ized MIDlet contains following attributes:

```
MIDlet-Name: Snake
MIDlet-Vendor: Motorola
MIDlet-Version: 1.0.0
MIDlet-Jar-URL: Snake.jar
MIDlet-Jar-Size: 5000
MIDlet-1: Snake, , com.motorola.snake.Snake

iDEN-MIDlet-Name-ko: 스네이크
iDEN-MIDlet-Vendor-ko: 모토로라
iDEN-MIDlet-ko-1: 스네이크, , com.motorola.snake.Snake

iDEN-MIDlet-Name-es: Serpiente
iDEN-MIDlet-Vendor-es: Motorola
iDEN-MIDlet-es-1: Serpiente, , com.motorola.snake.Snake

Gray   - MIDP Specification
Green  - iDEN Korean Extensions
Yellow - iDEN Spanish Extensions
```

The format of the JAD file is a sequence of lines consisting of an attribute name followed by a colon, the value of the attribute, and a carriage return. The attributes `iDEN-MIDlet-Name-ko`, `iDEN-MIDlet-Vendor-ko`, and `iDEN-MIDlet-ko-1` display the MIDlet suite, vendor, and friendly names in Korean accordingly. Language specific suite name, vendor name, and MIDlet name tags will be used when the phone's language setting matches specified language attributes in the JAD file. If special language attributes are not specified in the JAD file, the phone will use the default English MIDlet suite, vendor, and friendly names.

Manifest files remain in ASCII format and must follow the specifications in MIDP 1.0.

4. MIDP 2.0 LCDUI

4.1. Overview

With the changes in the keypad layout and the new MIDP 2.0 UI specification, developers must consider a few implementation specifics that may affect the look and feel of the applications. Although every effort has been made to support the backward compatibility of the applications, the numerous hardware and software specification required some changes in convention from previous handsets. The next few sections outline the implementation specifics that affect application layout and usability.

4.2. Commands

4.2.1. Layout Priorities

In the LCDUI specification, applications that contain multiple soft keys have the option of specifying priority in layout.

4.2.1.1. Previous Implementation

With the previous handsets, the highest priority commands appear above the right soft key with lower priority keys occupying the left. With only two dedicated keys, additional soft keys were added to a submenu, leaving the dedicated Menu key unutilized. If multiple commands contain the same priority level, the keys are displayed in the order they are added starting from the right soft key.

4.2.1.2. New Implementation

Although the numbers of keys have remained the same, changes were required in the implementation to better match the rest of the handset up. The soft key priority is as follows: left soft key, right soft key, and submenu. With only two dedicated keys, additional soft keys are added to a submenu, but are now accessed with the Menu key. If multiple commands contain the same priority level, the keys are assigned to the dedicated soft keys in the order they are added, following the priority convention.

Scheme	Commands with different priorities
i325	Left soft key, Right soft key, sub-menu.
i95cl and previous	Right soft key, Left soft key, sub-menu.
Note- Commands with same priorities are assigned keys in the order added, following the conventions in this table.	

4.3. Empty String Labels

The new look and feel of the UI includes changes to the visual appearance of the commands rendered. Many of the changes are direct results of efforts to better integrate the Java platform into other handset components.

4.3.1.1. Previous Implementation

The labels assigned to commands appeared in all instances as plain text followed by a blank background. Depending upon the handset, the background and font colors change according to the native color palette chosen.

4.3.1.2. New Implementation

One of the major improvements in the UI implementation for this release is native look and feel. This encompasses not only the color palette choice, but also the font, icons, and other various design elements. With regards to commands, and in addition to the color palette matching, an outline of the soft key area is now rendered along with the text. This places hard boundaries to the areas rendered by the platform. If a command is created with an empty string (""), the command is not rendered but still occupies the soft key location. Additionally, if the soft key is activated by the user, the `commandAction()` method is still called. This is useful for placing commands in explicit locations, a feature not available with the standard LCDUI. If a command is created with a short string (" ") , it will be rendered on the display with no visible font.

4.3.2. Short and Long Label Usage

A major shortcoming of the previous LCDUI prevented commands from specifying varying lengths of the labels. This resulted in command text being truncated depending on where it was rendered. The problem lies specifically with the inability of an application to determine the location the commands are rendered. The ability for a command to assign different length labels alleviates this issue while remaining hardware independent and backwards compatible.

4.3.2.1. Implementation Specifics

On the i325, the short strings are used if the command is placed on a dedicated soft key. In the command submenu, long strings are rendered if specified. If no long string is present, the short string is used instead.

Note – Even with the new short/long string feature, instances where truncation is necessary will arise. The i325 will only display full characters (the trailing characters are truncated) with labels justified according to the language.

4.4. Canvas

4.4.1. Size Changes

A major shortcoming of the previous Canvas spec was the inconsistency in which command/canvas interactions were treated. Most implementations either reserved space for the commands or gave the applications full screen access. This inconsistency resulted in commands overwriting application screen real estate. Per the new LCDUI, canvas based applications are now able to accommodate for screen size changes, regardless of the implementation.

4.4.1.1. Previous Implementation

Canvas sizes remained constant regardless of commands or other platform components. This behavior resulted in platform components obscuring and overwriting application components. The typical workaround entailed hard coding screen and font dimensions in the application.

4.4.1.2. New Implementation

Any platform components that occupy real estate and are added to the canvas results in a call to the `sizeChanged()` method, followed by a paint. Beyond adding and removing commands, addition or removal of tickers or titles invokes a call to `sizeChanged()`. Within the `sizeChanged()` callback, applications should query the new canvas size and adjust rendering accordingly. The following table summarized the conditions that trigger the callback:

Conditions	Notes
Adding/Removing Commands to Canvas	Only first command added triggers `sizeChanged()`. Only last command removed triggers `sizeChanged()`.
Adding/Removing Ticker to Canvas	
Adding/Removing Title to Canvas	
Call to `setFullScreenMode()`	
Change in font size*	Changes in ergo settings menu applies if com.motorola.iden.lnf package is utilized to `getDefaultFont()`.

4.5. List

4.5.1. No OK Key

For the i325 platform and other derivative products, There is no physical OK key is mapped to the default `SELECT_COMMAND` in javax.microedition.lcdui.List. Applications utilizing List should listen for the command in `commandAction()`. For javax.microedition.lcdui.ChoiceGroup, the physical Send key operates as the select key for `EXCLUSIVE`, `MULTIPLE`, and `POPUP` types.

4.5.2. Fit Policy

In MIDP 2.0, the application is given an option to specify Choice fit policy within a List. The functionality is optional per spec and is present only to provide hints to the platform as to the desired layout. The application cannot rely on the availability of the fit policy. The i325 does not support this functionality, but the API does exist for compatibility's sake.

Note – The default fit policy is wrapping.

4.6. Forms

4.6.1. Item Layout

Considerable layout directives changes for Items have been incorporated in the new spec. In the previous spec, layout directives applied strictly to ImageItems within Forms. Examples of such directives include `LAYOUT_NEWLINE_BEFORE`, `LAYOUT_CENTER`, and `LAYOUT_DEFAULT`. These directives provide layout hints to the platform as to how ImageItems are arranged within a Form. For MIDP 2.0, the scope of layout directives has been broadened to incorporate StringItem, CustomItem, and Spacer as well as ImageItem. In addition to expanding the reach of this functionality, additional layout directives have been added to increase the flexibility and usefulness.

4.6.1.1. Previous Implementation

Items added or appended to a Form occupy a new row, regardless of the available space. This particular layout semantic provided for easy and simple implementation of platform and application. The downfall of a simple scheme becomes apparent when complex and mixed Item types are added to a Form. For example, mixing text along with images lead to large gaps of unutilized space.

4.6.1.2. New Implementation

As stated earlier, the effectiveness of the layout directives have been moved from ImageItem, up to the superclass Item. Any subclass of Item is now capable of storing its own layout directive. In addition to broadening the reach, additional directives such as `LAYOUT_LEFT`, `LAYOUT_RIGHT`, and `LAYOUT_TOP` are included. By default, if no directive is specified, new Items added to the Form inherit the layout directives of the previous Item. By default, if Items do not specify a layout directive, they are added row by row, from left to right.

Note - StringItems are also appended in this same manner, differing from the previous implementation. For more information regarding layout changes, please refer to JSR 118. The following table outlines the behavior of Items that do not follow the standard layout directives:

Item	Default Behavior
Gauge	Center Justified.
ChoiceGroup	Language Dependent – default left justified.
DateField	Language Dependent – default left justified.
TextField	Screen width – default caret position is language dependent.

4.6.2. Item Commands

Utilizing a platform that resides on devices with very limited input and output mechanisms poses many challenges for application developers. In providing a myriad of options to the user, applications must present the user with a simple UI structure while still providing all the functionalities advanced users require. With the previous implementation of LCDUI, commands are constrained to Forms exclusively. That is, commands can only be added to Forms. While this is sufficient for simple applications, the sophistication and complexity of today's applications are quickly outgrowing this model. For MIDP 2.0, commands reside not only in Forms, but may also be added to individual Items.

4.6.2.1. Previous Implementation

Commands are restricted to Displayable objects (i.e. Screens and Canvas) exclusively. Options for all the Items are collectively grouped and presented together in a command submenu. This quickly leads to an overload of options, many of which do not apply to the Item of interest.

4.6.2.2. New Implementation

Commands may be added to individual Items instead of exclusively to the Displayable. For the i325, the availability of these commands is conveyed to the user when the Item is highlighted. If a command is available to a highlighted item, the Menu icon is displayed. Pressing the Menu key brings the command submenu to the foreground. If no commands are available to a highlighted item, the Menu icon is not shown. If, however, the Form contains more then 2 commands, the Menu icon will always be highlighted. The following table summarizes the characteristics.

Condition	Menu Icon	Soft keys
0 Displayable Commands 1 Item Commands	Yes	None
2 Displayable Commands 0 Item Commands	No	Displayable Commands
2 Displayable Commands 1 Item Command	Yes	Displayable Commands
Note – If the Displayable contains more then 2 commands, the MENU icon will always be displayed. There is no means for the end-user to determine if Items have commands. Avoid adding more then 2 commands to the Displayable.		

4.7. TextBox/ TextField

4.7.1. Commands and Edit Mode Icons

The Command changes listed above, along with the i325's new native UI have also changed the way text entry notification is displayed for MIDlets on the i325. When a MIDlet's screen is focused on a TextField or a TextBox the current entry mode (Alpha, Word, or Numeric) is displayed in the bottom, middle of the screen (between the Command labels, if present). This icon not only indicates entry mode but shift state, as well. However, if the screen has more than two Commands associated with it, or if the TextField has an ItemCommand, the entry mode icon will be replaced by the Menu icon. In this case a user will not be able to easily tell what editing mode or shift state they are in and will be required to cycle through Command menus before being able to change the entry mode. If possible, developers are encouraged to avoid using ItemCommand with TextField and to keep TextFields on screens with two or less Commands.

4.7.2. Constraints and Initial Input Modes

Text entry using the standard ITU keypad is not only cumbersome, but error prone. To alleviate this, many manufacturers incorporated predictive text entry of one kind or another to ease the burden. The group for JSR 118 has taken this into consideration and incorporated multiple input modes within the high-level LCDUI components. First and foremost, the value constraints from MIDP 1.0 remain, including `ANY`, `PHONENUMBER`, and `URL`,. In addition, new modifier flag constraints have been added to let the developer control the smart text engine. The i325 honors all of the modifier flag constraints: `PASSWORD`, `UNEDITABLE`, `SENSITIVE`, `NON_PREDICTIVE`, `INITIAL_CAPS_WORD`, and `INITIAL_CAPS_SENTENCE`.

MIDP 2.0 also introduces the concept of input modes for even finer control of text components. The input mode is simply a request for a specific set of characters to be entered more conveniently. Since not all devices and platforms will support all modes, no specific input modes are required by the MIDP 2.0 specification. The i325 supports several of the suggested input modes as well as some platform specific additions. The following table lists the supported input modes:

Constraint	Description
MIDP_UPPERCASE_LATIN	Defined by MIDP, this input mode turns on caps lock and switches the text component to English if the text component is currently in a non-Latin language.
MIDP_LOWERCASE_LATIN	Defined by MIDP, this input mode turns off caps lock or character shifting and switches the text component to English if the text component is currently in a non-Latin language.
IS_LATIN_DIGITS	Defined by J2SE™, this input mode switches the text component to NUMERIC mode if necessary.
UCB_BASIC_LATIN	Defined by J2SE™, this Unicode character block subset input mode switches the text component to English if the text component is currently in a non-Latin language. This input mode is equivalent to UCB_LATIN-1_SUPPLEMENT.
UCB_LATIN-1_SUPPLEMENT	Defined by J2SE™, this Unicode character block subset input mode switches the text component to English if the text component is currently in a non-Latin language. This input mode is equivalent to UCB_BASIC_LATIN.
UCB_HEBREW	Defined by J2SE™, this Unicode character block subset input mode switches the text component to Hebrew language mode if necessary and if the phone is configured for Hebrew support.
UCB_HANGUL_SYLLABLES	Defined by J2SE™, this Unicode character block subset input mode switches the text component to Korean language mode if necessary and if the phone is configured for Korean support.
X_MOTOROLA_IDEN_ENGLISH	Defined specifically for the i325, this input mode ensures that the text component is in English language mode.
X_MOTOROLA_IDEN_SPANISH	Defined specifically for the i325, this input mode ensures that the text component is in Spanish language mode.
X_MOTOROLA_IDEN_FRENCH	Defined specifically for the i325, this input mode ensures that the text component is in French language mode.
X_MOTOROLA_IDEN_PORTUGUESE	Defined specifically for the i325, this input mode ensures that the text component is in Portuguese language mode.

While the Motorola-defined input modes do allow developers to change the language setting for a particular text component, it is important to note that text component language is automatically selected depending on the phone's language setting. Manually forcing a specific language should be used with caution as it can create a bad user experience. In addition, well-written applications should store and reset user specified input modes and constraints between sessions.

5. MIPD 2.0 Push Registry

5.1. Overview

Push registration lets a MIDlet set itself to be launched automatically. The push registry allows for registering network and timing based activation and also manages the MIDlet activation process defined by MIDP 2.0 push registry.

The i325 implementation of push registry supports all methods defined in the MIDP 2.0 PushRegistry class.

5.2. Network Launch

The i325 implementation supports three network protocols: datagram (UDP), socket (TCP) and SMS. An application can be statically registered by defining a property in a descriptor file or it can register dynamically by calling the register connection API during run time. To register an application for static socket (TCP) connections, the device must have packet data service. To receive inbound messages, the device must have packet data or SMS service. This can require special provisioning by the carrier or service provider. The i325 supports a maximum of eight push registrations; a MIDlet may have multiple push registrations.

The i325 implementation buffers the first incoming datagram or SMS message before it launches the MIDlet. Once the MIDlet launches, the connection delivers this message, and all subsequent messages are delivered directly to the application. The MIDlet is of course responsible for opening the connection using the Generic Connection framework. For sockets, the MIDlet is launched when a TCP connection is established, and the connection is transferred to an application after it is launched.

The sections below describe device-specific information regarding registration on the i325. For more information on PushRegistry consult the MIDP 2.0 specification.

5.3. Time-based Launch

Time-based launch is accomplished using the `registerAlarm()` method detailed below. Each application only has access to one alarm and only one future event can be pending. The maximum number of alarms that are available at any one time is 32. An application is launched only if the phone is powered on. If the phone is powered off and an alarm goes off, the application will not be launched.

5.4. Class Description

The API for the PushRegistry is located in package javax.microedition.io.

```
java.lang.Object
 |
 + - javax.microedition.io.PushRegistry
```

5.5. Method Description

5.5.1. PushRegistry Method

5.5.1.1. registerAlarm

```
static long registerAlarm (String midlet, long time)
         throws ClassNotFoundException, ConnectionNotFoundException
```

You can delete a previously registered alarm by setting the time parameter to zero. The registered time must be local time. The time must be a minimum of two minutes in the future from the current time.

5.6. Tips

It's recommended that you open the connection immediately in a separate thread in the MIDlet's `startApp()` and read the received message.

Applications should not use any reserved ports as defined by IANA, for example FTP, Telnet, or HTTP.

6. MIDP 2.0 Record Management System (RMS)

6.1. Overview

The most common mechanism for persistently storing data on a MIDP device is through RMS. RMS lets a MIDlet store variable length records on the device. Those records are accessible to any MIDlet in the MIDlet suite, and also to MIDlets outside of the MIDlet suite if permission is given when the record is created. The RMS implementation on the i325 phone is MIDP 2.0 compliant.

MIDlets within a suite can access each other's record stores directly. New APIs in MIDP 2.0 let you explicitly share record stores if the MIDlet creating the record store chooses to give such permission. Sharing is accomplished through the ability to name a record store created by another MIDlet suite.

You define access controls when you create a record store that's to be shared. Access controls are enforced when RecordStores are opened. The access modes allow private use or shareable with any other MIDlet suites.

6.2. Class Description

The API for the RecordStore is located in the package javax.microedition.rms.

6.3. Code Examples

The following simple code example opens a record store. If any exception occurs it is caught.

```
try {
      System.out.println("Opening RecordStore " + rsName + " ...");

      //try to open a record Store
      recordStore = RecordStore.openRecordStore(rsName, true);

      //keep a note for the last modified time for record store
      Date d = new Date(recordStore.getLastModified());
      System.out.println(recordStore.getName()+"modified last time: " +
      d.toString());
}
catch (RecordStoreException rse) {
      //process the IOException
}
```

The following simple code example will open (and possibly create) a record store that can be shared with other MIDlet suites. The record store is owned by the current MIDlet suite. The authorization mode is set when the record store is created, as follows:

`AUTHMODE_PRIVATE` allows only the MIDlet suite that created the record store to access it. This case behaves identically to `openRecordStore(recordStoreName, createIfNecessary).`

`AUTHMODE_ANY` allows any MIDlet to access the record store. Note that this makes your record store accessible by any other MIDlet on the device. This could have privacy and security issues depending on the data being shared. Please use carefully.

```
try {
   System.out.println("Opening RecordStore " + rsName + " ...");

   //try to open a record store
   recordStore = RecordStore.openRecordStore(rsName,true,
      (byte)RecordStore.AUTHMODE_ANY, true);

   //keep a note for the last modified time for record store
   Date d = new Date(recordStore.getLastModified());
   System.out.println(recordStore.getName()+"modified last time: " +
      d.toString());
} catch (RecordStoreException rse) {
   //process the IOException
}
```

6.4. Tips

It is much faster to read and write in big chunks than it is to do so in small chunks. The optimal size for reading and writing is 512 bytes.

Whenever you close a record store, `close()` does not return until all the pending writes have been written. A successful `close()` call guarantees that the data was written. It is then safe to power off the phone. Because of this, `close()` may take a while to return. Therefore, if a record store is opened and closed for every write, performance will slow down greatly.

6.5. Caveats

The i325 phone supports a maximum of 2048 record stores. If there is no file space available, you cannot create extra record stores or records. Once the phone contains 2048 record stores, you cannot create more. MIDI ringers, voice notes, wallpapers, PNG images included with a MIDI are all files. If a MIDlet has many images, such as sprites used in animations, it may be advantageous to have them all in one image file and use clipping to display only what you need.

A record store can be of any size as long as there is file space available. A zero byte record store is also allowed.

Each MIDlet suite is guaranteed to be able to open at least 5 files or record stores simultaneously.

There is an additional pool of 16 files and record stores that can be opened. This pool is shared among all MIDlet suites, giving a MIDlet suite the potential to simultaneously open 21 files or record stores.

6.6. Compiling and Testing RMS MIDlets

This is a standard MIDP 2.0 package so there is no need for stub classes to compile the MIDlet with RMS APIs.

7. MIDP 2.0 File I/O and Secure File I/O

7.1. Overview

The objective of the File I/O and secure File I/O API is to provide a generic platform for the Java developer to use to open, read, write, append and delete a file sequentially. The goal is to provide UNIX like file access APIs, as a simple alternative to Record Management System (RMS). This lets MIDlets save information between invocations; this is called "persistent storage." Examples include:

Saving data such as notes, phone numbers, tasks, and so on.

Keeping a history of recent URLs

Secure File I/O API provides a generic platform for the Java developer to protect the persistent storage with password protection.

7.2. Class Description

The File I/O and Secure File I/O APIs are located in package javax.microedition.io.

7.3. Method Description

7.3.1. Connector Method

7.3.1.1. open

Opens or deletes the specified file.

```
public static Connection open(String  name)
   throws IOException

public static Connection open(String name, int mode)
   throws IOException

public static Connection open(String  name, int mode,
   boolean timeouts) throws IOException
```

Opening a file gives your application exclusive access to that particular file until it is explicitly closed or the program ends. Opening a secure file gives your application password-protected access to that particular file until it is explicitly closed or the program ends.

`name` is a URL that contains the name of the file to open, and can also include keywords that specify the mode in which to open it. Here are some examples:

- `"file://temp.txt"` specifies that file is to be opened in the default mode, which is `READ_WRITE`.
- `"file://temp.txt;APPEND"` specifies that file is to be opened in an `APPEND` mode.
- `"sfile://temp.txt;PASSWORD=313"` specifies that the file is a secure file to be opened with the password 313 and in the default mode, `READ_WRITE`.
- `"sfile://temp.txt;PASSWORD=313;APPEND"` specifies that the file is a secure file to be opened with the password 313 and in an `APPEND` mode.

You can also delete a file with the `DELETE` keyword. Note that all the InputStreams, OutputStreams and StreamConnections associated with a file should be closed before deleting the file. If a file cannot be deleted, these methods throw an IOException. Here are some examples of name parameters that delete a file:

- `"file://temp.txt;DELETE"` deletes temp.txt.
- `"sfile://temp.txt;PASSWORD=313;DELETE"` deletes the secure file temp.txt with the password 313.

`mode`, if included, must have one of these three values: `READ`, `WRITE`, or `READ_WRITE`.

`timeout` has no effect on the method call and is ignored.

These are five basic steps for reading and writing a file:

- Open the file using the `open()` method of Connector class. This returns a StreamConnection object for file. Otherwise, an IOException is thrown.
- Get the output stream using the `openOutputStream()` method of OutputConnection.
- Get the input stream using the `openInputStream()` method of InputConnection.
- Once the connection has been established, simply use the normal methods of any input or output stream to read and write data.
- Close the file using the `close()` method of Connection.

For more information, see the Javadocs.

7.4. Code Examples

7.4.1. Example # 1 (File/Secure File Snippet)

The following example shows how to open a file, write bytes to the file and read the same number of bytes.

```
StreamConnection sc = null;
InputStream is = null;
OutputStream os = null;

//For regular file
String name = "file://temp.txt";

//For secure file
String name = "sfile://temp.txt;PASSWORD=4509";

try {
     // open a file, default mode: READ_WRITE
     sc = (StreamConnection)Connector.open(name);

     // get OutputStream
     os = sc.openOutputStream();


     // get InputStream
     is = sc.openInputStream();
     String b = "Hello World";
```

```
      // write the bytes
      os.write(b.getBytes());
      int dataAvailable = is.available();
      byte [] b1 = new byte[dataAvailable];

      // read the bytes
      is.read(b1);
} finally {
       if (sc != null)
           sc.close();
       if (is != null)
           is.close();
       if (os != null)
           os.close();
}
```

7.4.2. Example # 2 (Complete File MIDlet Code)

The following example is a simple MIDlet that will provide the overall operation of the file I/O interface and how most of the API's can be used. The MIDlet also shows a simple alternative to RMS to store data as a persistent storage.

```
import javax.microedition.io.*;
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class Example2 extends MIDlet implements CommandListener{
      /**
       * List of available tests
       */
      StreamConnection sc;
      String[] testList = {"file to w/r", "setData",
            "write/append/read", "delete"};
      TextBox tf1;
      TextBox tf2;

      /**
       * Reference to Display object associated with this Display
       */
      Display myDisplay;

      /* default file name */
      String fileURL = "temp.txt";

      /*default amount of data*/
      int dataNum = 0;


      /*default string to write in file*/
      String stringNum ="Hello World";

      /**
       * The output screen
       */
      Form myOutput;
```

```
/**
 * The list of tests
 */
List myList;

/**
 * Ok command to indicate a test was selected
 */
Command okCommand;

/**
 * Create NetTests
 */

public Example2( ) {

}

/**
 * Start running
 */
    protected void startApp() {
    myDisplay = Display.getDisplay(this);
    myOutput = new Form("Results");
    myList = new List("Select test:", List.IMPLICIT, testList,
                null);
    okCommand = new Command("OK", Command.OK, 1);
    myOutput.addCommand(okCommand);
    myList.addCommand(okCommand);
    myOutput.setCommandListener(this);
    myList.setCommandListener(this);
    tf1 = new TextBox("file to w/r", fileURL, 28, TextField.ANY);
    tf1.addCommand(okCommand);
    tf1.setCommandListener(this);
    tf2 = new TextBox("Set Data to Send", stringNum, 28,
        TextField.ANY);
    tf2.addCommand(okCommand);
    tf2.setCommandListener(this);
    myDisplay.setCurrent(myList);
}

    /**
     * Stop running
     */
    protected void pauseApp() {
    }

    /**
     * Destroy App
     */
    protected void destroyApp(boolean unconditional) {
    }
```

```
/**
 * Handle ok command
 */
public void commandAction(Command c, Displayable s) {
    if (((s == tf1) || (s == tf2)) && (c == okCommand)) {
          if(s==tf1) fileURL = tf1.getString();
          if(s==tf2) {
           /* data in the string form */
            stringNum = tf2.getString();
           /* convert the string into the integer form */
            dataNum = stringNum.length();
          }
    }
    if (s == myList) {
        switch (((List)s).getSelectedIndex()) {
        case 0:
            myDisplay.setCurrent(myOutput);
            setFileName();
            break;
        case 1:
            myDisplay.setCurrent(myOutput);
            setData();
            break;
        case 2:
            myDisplay.setCurrent(myOutput);
            readWrite();
            break;
        case 3:
            myDisplay.setCurrent(myOutput);
            deleteFile();
            break;
        }
    } else {
        myDisplay.setCurrent(myList);
    }
}
private void setFileName() {
    myDisplay.setCurrent(tf1);
}

private void setData() {
      myDisplay.setCurrent(tf2);
}

private void readWrite() {
    int length = dataNum;
    byte[] message = new byte[length];
    message = stringNum.getBytes();
    OutputStream os = null;
    InputStream is = null;
    try {
        //open a file in the mode APPEND
        sc = (StreamConnection)Connector.open("file://" +
              "fileURL" + ";" + "APPEND");
```

```
            //get OutputStream
            os = sc.openOutputStream();

            //get InputStream
            is = sc.openInputStream();

            //write the bytes to the file
            os.write(message);
            myOutput.append("write/append done");

            //create an array to store available data
            // from the file
            byte [ ] b1 = new byte[is.available()];

            //read the bytes
            is.read(b1);
            String readString = new String(b1);

            //printout the data in the phone screen
            myOutput.append(readString);
            myOutput.append("read finished");

            //close all the opened streams
            if (is != null)
                is.close();
            if (os != null)
                os.close();
            if (sc != null)
                sc.close();
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());

            try {
                if (is != null)
                    is.close();
                if (os != null)
                    os.close();
                if (sc != null)
                    sc.close();
            }
            catch(Exception ex) {
            }
          }
    }
    private void deleteFile() {
        try {

            //open a file in the delete mode
            //by doing this existing file is eventually delete
            sc = (StreamConnection)Connector.open("file://" +
                  "fileURL" + ";" + "DELETE");
```

```
                //close the stream
                if (sc != null)
                    sc.close();
                myOutput.append("file deleted");
            } catch (Exception ex1 ) {
                System.out.println("Exception: " + ex1.getMessage());
                try {
                    if (sc != null)
                        sc.close();
                }
                catch(Exception ex) {
                }
            }
        }
}
```

7.4.3. Example # 4 (Complete Secure File MIDlet Code)

The following example is a simple MIDlet that provides the overall operation of the secure file I/O interface and how most of the API's can be used. The MIDlet also shows a simple alternative to RMS to store data as a persistent storage with password protection.

```
import javax.microedition.io.*;
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class Example4 extends MIDlet implements CommandListener{

/**
 * List of available tests
 */
StreamConnection sc;
String[] testList = {"file to w/r","setData","write/append/read","delete"};
TextBox tf1;
TextBox tf2;

/**
 * Reference to Display object associated with this Display
 */
Display myDisplay;

/* default file name */
String fileURL = "temp.txt;PASSWORD=1413";

/*default amount of data*/
int dataNum = 0;

/*default string to write in file*/
String stringNum ="Hello World";

/**
 * The output screen
 */
Form myOutput;
```

```
/**
 * The list of tests
 */
List myList;

/**
 * Ok command to indicate a test was selected
 */
Command okCommand;

/**
 * Create NetTests
 */
public Example4( ) {

}

/**
 * Start running
 */
    protected void startApp() {
    myDisplay = Display.getDisplay(this);
    myOutput = new Form("Results");
    myList = new List("Select test:", List.IMPLICIT, testList,
                null);
    okCommand = new Command("OK", Command.OK, 1);
    myOutput.addCommand(okCommand);
    myList.addCommand(okCommand);
    myOutput.setCommandListener(this);
    myList.setCommandListener(this);
    tf1 = new TextBox("file to w/r", fileURL, 28, TextField.ANY);
    tf1.addCommand(okCommand);
    tf1.setCommandListener(this);
    tf2 = new TextBox("Set Data to Send", stringNum, 28,
                      TextField.ANY);
    tf2.addCommand(okCommand);
    tf2.setCommandListener(this);
    myDisplay.setCurrent(myList);
 }

    /**
     * Stop running
     */
    protected void pauseApp() {
    }

    /**
     * Destroy App
     */
    protected void destroyApp(boolean unconditional) {
    }
```

```
/**
 * Handle ok command
 */
public void commandAction(Command c, Displayable s) {
    if (((s == tf1) || (s == tf2)) && (c == okCommand)) {
       if(s==tf1) fileURL = tf1.getString();
       if(s==tf2) {
            /* data in the string form */
            stringNum = tf2.getString();
            /* convert the string into the integer form */
           dataNum = stringNum.length();
       }
    }
    if (s == myList) {
        switch (((List)s).getSelectedIndex()) {
        case 0:
            myDisplay.setCurrent(myOutput);
            setFileName();
            break;
        case 1:
            myDisplay.setCurrent(myOutput);
            setData();
            break;
        case 2:
            myDisplay.setCurrent(myOutput);
            readWrite();
            break;
        case 3:
            myDisplay.setCurrent(myOutput);
            deleteFile();
            break;
        }
    } else {
        myDisplay.setCurrent(myList);
    }
}
private void setFileName() {
    myDisplay.setCurrent(tf1);
}
private void setData() {
    myDisplay.setCurrent(tf2);
}
private void readWrite() {
    int length = dataNum;
    byte[] message = new byte[length];
    message = stringNum.getBytes();
    OutputStream os = null;
    InputStream is = null;
    try {
        //open a file in the mode APPEND
        sc = (StreamConnection)Connector.open("sfile://" +
              fileURL + ";" + "APPEND");

        //get OutputStream
        os = sc.openOutputStream();
```

```
            //get InputStream
            is = sc.openInputStream();

            //write the bytes to the file
            os.write(message);
            myOutput.append("write/append done");

            //create an array to store available data from the file
            byte [ ] b1 = new byte[is.available()];

            //read the bytes
            is.read(b1);
            String readString = new String(b1);

            //printout the data in the phone screen
            myOutput.append(readString);
            myOutput.append("read finished");

            //close all the opened streams

            if (is != null)
                is.close();
            if (os != null)
                os.close();
            if (sc != null)
                sc.close();
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            try {
                if (is != null)
                    is.close();
                if (os != null)
                    os.close();
                if (sc != null)
                    sc.close();
            }
            catch(Exception ex) {
            }
          }
    }

    private void deleteFile() {
        try {
            //open a file in the delete mode
            //by doing this existing file is eventually delete
            sc = (StreamConnection)Connector.open("sfile://" +
                  fileURL + ";" + "DELETE");

            //close the stream
            if (sc != null)
                sc.close();
            myOutput.append("file deleted");
        } catch (Exception ex1 ) {
            System.out.println("Exception: " + ex1.getMessage());
```

```
                try {
                    if (sc != null)
                        sc.close();
                }
                catch(Exception ex) {
                }
            }
        }
    }
```

7.5. Tips

Like RMS, it is much faster to read and write in big chunks than it is to do so in small chunks. The optimal size for reading and writing is 512 bytes.

File I/O is a simple alternative to Record Management System (RMS). When used effectively, direct file I/O can speed up storing and retrieving data.

After creating a file from a MIDlet suite, the file is associated with the current MIDlet suite only. No other MIDlet suite can access it.

If a MIDlet suite is updated to another version, then the file(s) associated with the current version of MIDlet suite can be maintained for the new version to use. The user is prompted to keep the old data or delete it.

If a MIDlet suite is deleted, all files associated with it are deleted.

It is a MIDlet's responsibility to coordinate the use of multiple threads to access a file since unintended consequences may result.

Whenever you close a file, the `close()` command will not return until all the pending writes have been completed; thus closing a file guarantees that all of the data is written. It is then safe to power off the device. One consequence is that the `close()` command may take a while to return. Therefore, if you open and close the file every time a write is required, performance will be greatly affected.

Secure File I/O API has all the functionality of regular File I/O, but in addition it provides password protection to the persistent storage.

7.6. Caveats

This File Access System is a sequential system. This means once you write particular chunk of data to the file, you can't go back and manipulate it.

Theoretically, the maximum number of files that the i325 phone supports is 2048. If there is no file space available, one cannot create extra files. And once the phone contains 2048 files, it will not be able to create more. MIDI ringers, voice notes, wallpapers, PNG images included with a MIDlet are all files. If a MIDlet has many images, such as sprites used in animations, it may be advantageous to have them all in one image file and use clipping to display only what you need.

The file name can contain up to 32 alphanumeric 32 characters.

A file can be of any size as long as file space is available.

A zero-byte file is not allowed. Unwanted behavior may occur when a file is opened and no byte is written in to it before closing it.

It is recommended that only 21 files remain open at one time. Exceeding the maximum number of opened files can result in unintended behavior.

The InputStream method `markSupported()` returns true only if the file open mode is `READ` or `APPEND`. This means that in any other file open mode, the `mark()` and `reset()` methods do not work.

In the InputStream method `mark()`, the `readlimit` argument tells the input stream to allow that many bytes to be read before the mark position gets invalidated. Since this operation is on a file, "remembering" the entire contents of stream/file does not incur any type of cost, so the `readlimit` parameter is ignored, preventing mark position invalidation.

A secure file can only be opened with the correct password. A wrong password cannot open the file and will throw an exception.

A password can have length up to 32 alphanumeric characters.

7.7. Compiling and Testing File/Secure File MIDlets

The file I/O APIs and secure file I/O APIs are based off of generic `Connector.Open()` APIs, so there is no need of any stub classes to compile the MIDlet with these APIs

8. MIDP 2.0 Security API

8.1. Overview

The MIDP 2.0 Security API consists of HttpsConnection, SecureConnection, SecurityInfo, Certificate and CertificateExecption.

The HttpsConnection class defines the necessary methods and constants to establish a secure network connection. The URL that specifies `HTTPS` when passed to `Connector.open` will return an `HttpsConnection`.

The SecureConnection class defines the secure socket stream connection. A secure connection is established using `Connector.open()` and a URL that specifies SSL. The secure connection is established before the `open` method returns. If the secure connection cannot be established due to errors related to certificates, a CertificateException is thrown. A secure socket is accessed using a generic connection string with an explicit host and port number. The host may be specified as a fully qualified host name or IPv4 number. For example, `ssl://host.com:79` defines a target socket on the `host.com` system at port `79`. Note that RFC1900 recommends the use of names rather than IP numbers for best results in the event of IP number reassignment.

The SecurityInfo class defines methods to access information about a secure network connection. Protocols that implement secure connections may use this interface to report the security parameters of the connection. It provides the certificate, protocol, version, and cipher suite, etc. in use.

Certificates are used to authenticate information for secure Connections. The Certificate interface provides to the application information about the origin and type of the certificate.

The CertificateException encapsulates an error that occurred while a Certificate is being used. If multiple errors are found within a Certificate the more significant error should be reported in the exception.

8.2. Class Descriptions

The API for the HttpsConnection, SecureConnection, and SecurityInfo is located in package java.microedition.io. The API for the Certificate and CertificateException is located in package java.microedition.pki.

```
java.lang.Object
 |
 + - java.microedition.io.HttpsConnection
 |
 + - java.microedition.io.SecureConnection
 |
 + - java.microedition.io.SecurityInfo
 |
 + - java.microedition.pki.Certificate
 |
 + - java.microedition.pki.CertificateException
```

8.3. Method Descriptions

Please refer to MIDP 2.0 Javadocs.

8.4. Code Examples

8.4.1. HttpsConnection

The following is the code example of HttpsConnection: open a HTTPS connection, set its parameters, then read the HTTP response.

```
void getViaHttpsConnection (String url)
            throws CertificateException, IOException
{
         HttpsConnection c = null;
         InputStream is = null;
         try {
             c = (HttpsConnection) Connector.open(url);

             // Getting the InputStream ensures that the connection
             // is opened (if it was not already handled by
             // Connector.open()) and the SSL handshake is exchanged,
             // and the HTTP response headers are read.
             // These are stored until requested.
             is = c.openDataInputStream();

             if (c.getResponseCode() == HttpConnection.HTTP_OK)
            {
                 // Get the length and process the data
                 int len = (int)c.getLength();
                 if (len > 0)
                 {
                     byte[] data = new byte[len];
                     int actual = is.readFully(data);
                     ...
                 } else {
                     int ch;
                     while ((ch = is.read()) != -1) {
                         ...
                     }
                 }
             } else {
               ...
             }
         } finally {
             if (is != null)
                 is.close();
             if (c != null)
                 c.close();
         }
     }
```

8.4.2. SecureConnection

The following examples show how a `SecureConnection` would be used to access a sample loopback program.

```
SecureConnection sc = (SecureConnection)
Connector.open("ssl://host.com:79");
SecurityInfo info = sc.getSecurityInfo();
boolean isTLS = (info.getProtocolName().equals("TLS"));

sc.setSocketOption(SocketConnection.LINGER, 5);

InputStream is  = sc.openInputStream();
OutputStream os = sc.openOutputStream();

os.write("\r\n".getBytes());
int ch = 0;
while(ch != -1) {
    ch = is.read();
}

is.close();
os.close();
sc.close();
```

8.4.3. SecurityInfo

```
HttpsConnection c = null;
InputStream is = null;

c = (HttpsConnection) Connector.open("https://www.bellsouth.com/",
                                     Connector.READ_WRITE, true);
is = c.openInputStream();

try {
      secuInfo = c.getSecurityInfo();
} catch(Throwable t) {
      t.printStackTrace();
}

System.out.println(" ProtocolVersion "+secuInfo.getProtocolVersion());
System.out.println(" ProtocolName " + secuInfo.getProtocolName());
System.out.println(" CipherSuite " + secuInfo.getCipherSuite());
```

8.4.4. Certificate

```
Certificate cer = secuInfo.getServerCertificate();

System.out.println(" CA Type " + cer.getType());
System.out.println(" CA Version " + cer.getVersion());
System.out.println(" CA NotAfter " + cer.getNotAfter());
System.out.println(" CA NotBefore " + cer.getNotBefore());
System.out.println(" CA Subject " + cer.getSubject());
System.out.println(" CA Issuer " + cer.getIssuer());
System.out.println(" CA SerialNumber " + cer.getSerialNumber());
```

8.4.5. CertificateExecption

```
try {
      c = (HttpsConnection)Connector.open("https://www.bellsouth.com/",
                              Connector.READ_WRITE, true);
      is = c.openInputStream();
      . . . . .


} catch (CertificateException ce) {
      System.out.println ("Unexpected CertificateException  " + ce);
}
```

8.5. Tips

- HTTPS is the secure version of HTTP (IETF RFC2616), a request-response protocol in which the parameters of the request must be set before the request is sent.
- In addition to the normal IOExceptions that may occur during invocation of the various methods that cause a transition to the Connected state, CertificateException (a subtype of IOException) may be thrown to indicate various failures related to establishing the secure link. The secure link is necessary in the Connected state so the headers can be sent securely. The secure link may be established as early as the invocation of `Connector.open()` and related methods for opening input and output streams and failure related to certificate exceptions may be reported.
- MIDP 2.0 devices are expected to operate using standard Internet and wireless protocols and techniques for transport and security. The current mechanisms for securing Internet content is based on existing Internet standards for public key cryptography:
    - [RFC2437] - PKCS #1 RSA Encryption Version 2.0
    - [RFC2459] - Internet X.509 Public Key Infrastructure
    - [WAPCERT] - WAP-211-WAPCert-20010522-a - WAP Certificate Profile Specification
- On i325, only Verisign server certificates are supported. Using other server certificates will cause a CertificateException to be thrown.

9. MIDP 2.0 Platform Request

9.1. Overview

The Platform Request API allows a Java application to pass a URL to the phone to have it handled by one of the phone's native applications. The i325 Platform Request API supports only one type of URL: initiating a telephone call.

9.2. Class Description

The Platform Request API is located in package javax.microedition.midlet

```
java.lang.Object
 |
 + - javax.microedition.midlet.MIDlet
```

9.2.1. Method Description

9.2.1.1. platformRequest method

Passes a URL to the device to be handled be one of the phone's native applications.

```
public final boolean platformRequest(String URL)
   throws ConnectionNotFoundException
```

The URL must begin with either `"call:"` or `"tel:"`. If the URL begins with `"call:"`, the rest of the URL should contain a valid phone number. If the URL begins with `"tel:"`, the rest of the URL must be formatted according to RFC2806, which can be found at http://www.ietf.org/rfc/rfc2806.txt. When you pass this method a URL with `"call:"` or `"tel:"`, this method launches the native phone application, automatically entering the phone number from the URL. The user must press the Send key to complete the phone call.

9.3. Code Examples

```
public class platReq extends MIDlet implements CommandListener
{
    Display myDisplay;
    List myList;

    public void startApp() throws MIDletStateChangeException
    {
        myDisplay = Display.getDisplay(this);
        myList = new List("Select test:", List.IMPLICIT);

        myList.append("Call Test",null);
        myList.append("Tel Test",null);

        myList.append("Empty",null);
        myList.append("Invalid",null);
        myList.setCommandListener(this);
        myDisplay.setCurrent(myList);
    }
```

```
    public void pauseApp()
    {
    }

    public void destroyApp(Boolean unconditional)
    {
    }

    public void commandAction(Command c, Displayable s)
    {

        if (s == myList)
        {
          try
          {
             switch (((List)s).getSelectedIndex())
             {
               case 0:
                 platformRequest("call:5552313");
                 break;
               case 1:
                 platformRequest("tel:5552312;postd=10101010");
                 break;
               case 2:
                 platformRequest("");
                 break;
               case 3:
                 platformRequest("this is an invalid URL");
                 break;
             }
          }
          catch(Exception e)
          {
          }
        }
     }

}
```

9.4. Tips

- Once the native application receives the request, the application should be suspended and the user should be asked if he or she wants to follow through with the action.

10. Interconnect/Phone Call Initiation API

10.1. Overview

The Call Initiation API provides the ability to request an Interconnect service call. The API supports international and domestic phone numbers, "pause" dialing, and "wait" dialing.

The API does not actually make the call. Instead, it is designed to simply initiate a call request, wherefore then the end user must grant the request by pressing the Send key (also known as the Fire key in the Canvas Class). Since an application will be immediately suspended after calling the API, the employment of this interface must be from a separate thread other than the main thread. Upon successful call termination, the application will be resumed if the auto revert feature is enabled.

10.2. Class Description

The API for Call Initiation is located in package com.motorola.iden.call. The class, GenericCall, is the only class within the package and contains one static method to initiate the service calls. This class is a placeholder for the method:

```
java.lang.Object
 |
 + -- com.motorola.iden.call.GenericCall
```

10.3. Method Description

10.3.1. GenericCall Method

10.3.1.1. makeCall

Initiates a call request, which the user must grant by pressing the Send key.

```
public static int makeCall(String number) throws Exception
```

All MIDlet threads are kept running after calling this function and establishing the call. Packet data activity is stopped while in the phone call. Currently, you should start a call request only while the phone is idle (not active in any type of service call). Otherwise, this method throws an exception.

This method's argument specifies the number to dial. Its format is as follows:

```
number ::= [<Prefix Tag>] <Id>
```

The optional prefix tag included within the string is case insensitive and the Id (which has a maximum of 64 characters) determines the number dialed.

Table 4. `makeCall()` Argument Format

Prefix Tag	Id Value	Interconnect Call Behavior
"phon" or "PHON" none* *If the argument does not contain a tag, the request will be consider as an interconnect call.	**Domestic Interconnect Call**	
	(xxx)xxx-xxxx	-Dials xxxxxxxxxx
	Xxxxxxxxxx	-Dials xxxxxxxxxx
	(xxx)xxx-xxxxPyyy or (xxx)xxx-xxxx,yyy	-Dials xxxxxxxxxx, connects, and generates yyy DTMF's
	xxxxxxxxxxPyyPzzPPz or xxxxxxxxxx,yy,zz,,z	-Dials xxxxxxxxxx, connects, generates yy DTMF's, pauses for 3 seconds, generates zz DTMF's, pauses for 6 seconds, and generates z DTMF.
	xxxxxxxxxxWyy	-Dials xxxxxxxxxx and connects. Once the user presses the Send (or Fire) key, yy DTMF's are generated.
	International Interconnect Call	
	+(xxx)xxx-xxxx	-Dials xxxxxxxxxx with international "type of number"
	+xxxxxxxxxxPyy	-Dials xxxxxxxxxx with international "type of number", connects, and generates yy DTMF's.

To place a domestic call, the string argument may be "(XXX) XXX-XXXX", "XXXXXXXXX", "phonXXXXXXXXXX", or "PHONXXXXXXXXX". To make an international phone call, a '+' must be placed between the tag and the number to be dialed, as in "+XXXXXXXXX" or "phon+XXXXXXXXXX". To use pause dialing, insert a "P" (case sensitive) or ',' (Comma) as a pause digit. The first instance of the pause digit separates the phone number to be dialed from the DTMF tones that will be generated after the call is connected. Any subsequent "pause digit" inserts a 3-second break into the DTMF string at the specified location (refer to Table 1). To use wait dialing, insert a "W" (case sensitive) as the wait digit. The first instance of the wait digit separates the phone number to be dialed from the DTMF. The DTMFs are not generated until the user presses the Send (or Fire) key after each wait digit. The length of the ID portion of the argument used in `makeCall()` (that is, the length of the string excluding the tag) should not exceed 64 characters. Any number of blank spaces in the number is ignored. If the number is null or contains any invalid characters, this method throws an IllegalArgumentException.

Once in the dialing screen, the end user must grant the request by pressing the Send key (also known as the Fire key in the Canvas Class). If the auto-revert feature is enabled and the end user does not grant the request (i.e. press the Send key) within 3 seconds, the application is resumed.

The `makeCall()` method can successfully process only one request at a time. For example, if a thread makes a call request and is waiting for the response, any other thread will receive an exception when attempting to initiate a call. The exception will be thrown for each call request until the system is no longer busy (i.e. the first thread receives a response).

Return values of the `makeCall()` method are:

- `GenericCall.CALL_RESPONSE_OK` if phone call request is placed successfully (user must grant the request)
- `GenericCall.CALL_RESPONSE_FAILURE` if request fails because the unit is not idle (e.g. in an active service call such as a private call)
- `GenericCall.CALL_RESPONSE_ERROR` if an error occurred during request because the keypad is disabled and the application was attempting to dial a number not in the phonebook
- `GenericCall.CALL_UNKNOWN_ERROR` if an unknown error took place while requesting

10.4. Code Examples

```
void makePhoneCall() {
   try {
         String number = "9555555555";
                     int x = GenericCall.makeCall(number);
         if ( X == GenericCall.CALL_RESPONSE_OK) {
              // request to initiate has been successfully made
              // Note: does not mean phone call is finished

         }
         else {
             // Error occurred  while making request
         }
         }catch(IllegalArgumentException e){
   }
   catch(Exception e) {
   }
}
```

10.5. Compiling & Testing Interconnect Capable MIDlets

The stubbed GenericCall class is a non-functional class. The class is provided to build and run within any emulator. The class will make an attempt to display its method's behavior through `System.out` print statements when a method is called.

If the device is idle, `makeCall(9555555555)` displays the following:

```
Requesting to call to 9555555555
Waiting for request response
```
The MIDlet pauses here. After the user presses the Send key, the phone call is started and the MIDlet remains paused.
```
CALL_RESPONSE_OK
```

If the keypad is disabled and 955555555 is not in the phonebook, `makeCall(955555555)` displays the following:

```
Requesting to call to 955555555
CALL_RESPONSE_ERROR
```

If the device is busy in another call, `makeCall(955555555)` displays the following:

```
Requesting to call to 955555555
CALL_RESPONSE_FAILURE
```

11. RecentCalls API

11.1. Overview

The RecentCalls API lets you access the phone's recent calls data. It lets you read and remove recent call entries. However, it does not let you add to the recent calls list.

11.2. Class Descriptions

The API for the RecentCalls is located in package com.motorola.iden.recentcalls

Following is the class Hierarchy of RecentCalls API.

```
java.lang.Object
 |
 + - com.motorola.iden.recentcalls.RecentCalls
 |
 + - com.motorola.iden.recentcalls.RecentCallsEntry
```

Following is the Interface Hierarchy of the RecentCalls API.

```
com.motorola.iden.recentcalls.RCLListener
```

11.3. Method Descriptions

11.3.1. RecentCalls Methods

11.3.1.1. entryAt

Returns the RecentCallsEntry at the specified index.

```
public RecentCallsEntry entryAt(int index)
         throws IllegalArgumentException
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

`index` is the number of the RecentCallsEntry to return. Note that the first entry is at index 0.

If `index` is greater than the index for the last RecentCallsEntry, this method throws IllegalArgumentException.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.2. firstEntry

Returns the first RecentCallsEntry if the RecentCalls list is not empty, null if the list is empty

```
public RecentCallsEntry firstEntry()
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.3. lastEntry

Returns the last RecentCallsEntry if the RecentCalls list is not empty, null if the list is empty.

```
public RecentCallsEntry lastEntry()
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.4. refreshList

Synchronizes this RecentCalls list with the phone's native list of recent calls.

```
public boolean refreshList()
```

This method synchronizes the RecentCalls list with the phone's native list of recent calls, adding calls to the RecentCalls list that have been added to the native list and removing calls from the native list that have been removed from the RecentCalls list.

Be sure to call this method when the RecentCalls list is first created, or when you call `removeEntryAt()` or `removeAll()`. Additionally, it's a good idea to call this method before you access the RecentCalls list, to ensure that the list is up-to-date.

This method returns true if the operation was successful, false otherwise. If the operation fails, the list is left in the same condition it was in before the operation.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.5. capacity

Returns the maximum number of recent calls that this phone can store.

```
public int capacity()
```

11.3.1.6. doesContain

Returns true if the specified RecentCallsEntry is in this RecentCalls list; false, otherwise.

```
public boolean doesContain(RecentCallsEntry myEntry)
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.7. indexOf

Returns the index of the specified RecentCallsEntry.

```
public int indexOf(RecentCallsEntry myEntry)
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws SecurityException

11.3.1.8. currentUsage

Returns the number of RecentCallsEntries in this RecentCalls list.

```
public int currentUsage()
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.9. isListEmpty

Returns true if this RecentCalls list contains no entries; false, otherwise.

```
public boolean isListEmpty()
```

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.10. numIncomingCalls

Returns the number of incoming calls in this RecentCalls list.

```
public int numIncomingCalls()
```

If this RecentCalls list is empty, this method returns -1.

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.11. numOutgoingCalls

Returns the number of outgoing calls in this RecentCalls list.

```
public int numOutgoingCalls()
```

If this RecentCalls list is empty, this method returns -1.

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.12. numMissedCalls

Returns the number of missed calls in this RecentCalls list.

```
public int numMissedCalls()
```

If this RecentCalls list is empty, this method returns -1.

Be sure to call `refreshList()` before using this function, to ensure that the RecentCalls list is up-to-date.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.13. removeEntryAt

Deletes the RecentCallsEntry at the specified index from this RecentCalls list.

```
public boolean removeEntryAt(int entryNumber)
         throws IllegalArgumentException
```

This method returns true if the operation is successful, false otherwise. If the operation fails, check to see if the entry is still there and try again.

You must call `refreshList()` after calling this method, to ensure that the phone's native recent calls list matches this RecentCalls list.

`index` is the number of the RecentCallsEntry to remove. Note that the first entry is at index 0.

If `index` is greater than the index for the last RecentCallsEntry, this method throws an IllegalArgumentException.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.14. removeAll

Deletes every RecentCallsEntry in this RecentCalls list.

```
public boolean removeAll()
```

You must call `refreshList()` after calling this method, to ensure that the phone's native recent calls list matches this RecentCalls list.

If this application does not have the right permissions to read the native recent calls list, this method throws a SecurityException

11.3.1.15. setRCLListener

Set the recent calls listener to be the specified object.

```
public void setRCLListener(RCLListener l)
```

The recent call listener notifies the application when the phone's native recent calls list has changed by sending the specified object the `RCLActionListener()` method.

11.3.2. RecentCallsEntry Methods

11.3.2.1. getServiceCallType

Returns the service call type for this RecentCallsEntry.

```
public int getServiceCallType()
```

This method returns `JAVA_SERVICE_CALL_TYPE_PHONE`, `JAVA_SERVICE_CALL_TYPE_PRIVATE`, `JAVA_SERVICE_CALL_TYPE_TALKGROUP`, or `JAVA_SERVICE_CALL_TYPE_CALLALERT`.

11.3.2.2. getCallType

Returns the call type for this RecentCallsEntry.

```
public int getCallType()
```

If the service call type for this RecentCallsEntry is `JAVA_SERVICE_CALL_TYPE_PHONE`, this method returns `JAVA_CALL_TYPE_INCOMING`, `JAVA_CALL_TYPE_OUTGOING`, or `JAVA_CALL_TYPE_MISSED`. Otherwise, this method returns `CALL_TYPE_NO_STATUS`.

11.3.2.3. getPhoneNumber

Returns the phone number for this RecentCallsEntry.

```
public String getPhoneNumber()
```

11.3.2.4. getDuration

Returns the duration of this call in seconds.

```
public int getDuration()
```

11.3.2.5. getMinute

Returns the minute at which this call was made.

```
public int getMinute()
```

11.3.2.6. getHour

Returns the hour at which this call was made.

```
public int getHour()
```

11.3.2.7. getDay

Returns the day in which the call was made, as integer from 1 to 31.

```
public int getDay()
```

11.3.2.8. getMonth

Returns the month in which the call was made, as an integer from 1 to 12.

```
public int getMonth()
```

11.3.2.9. getStsMsg

Return the status message string associated with the specific alert.

```
public String getStsMsg ()
```

11.3.3. RCLListener Method

11.3.3.1. RCLActionListener

Called when the phone's native recent calls list is changed.

```
public void RCLActionListener()
```

You should implement this method to be notified when the RecentCalls list changes. There can be only one recent calls listener per MIDlet.

11.4. Code Examples

The following is the code example of RecentCalls API

```
RecentCalls RCL =new RecentCalls();
RecentCallsEntry myEntry = new RecentCallsEntry;

if(RCL.isListEmpty()){
   System.out.println("This Recent Calls List is empty");
} else {
   try {
         if(RCL.refreshList()){
             try {
               int currentUsage = RCL.currentUsage();
             } catch(Exception e) {
               System.out.println(
               "Exception thrown in currentUsage()" + e);
             }
             try {
               for(int i = 0; i < currentUsage; i++){

                   myEntry = RCL.entryAt(i);
                   System.out.println("Phone number for Entry " +
                           i + " is " + myEntry.getPhoneNumber());
                   System.out.println("Call Type for Entry " + i  +
                           " is " + myEntry.getCallType());
                   System.out.println("Service Call Type for Entry "
                           + i  + " is " + myEntry.getServiceCallType());
                   System.out.println(
                           "The time of the call  for Entry " + i  +
                           " is " + myEntry.getHour() + ":" +
                           myEntry.getMinute());
                   System.out.println(
                           "The date of the call  for Entry " + i  +
                           " is " + myEntry.getMonth() + "-" +
                           myEntry.getDay());
                   System.out.println(
                           "The duration of the call for Entry " +
                           i + " is " + myEntry.getDuration());
                   try{
                     if(RCL.doesContain(myEntry)){
                         System.out.println(
                           "doesContain() returned true");
                     }else {
                         System.out.println(
                           "doesContain() returned false");
                     }
                   } catch(Exception e) {
                     System.out.println(
                           "Exception thrown in doesContain()" + e);
                   }
               }
             } catch(Exception e) {
               System.out.println("Exception thrown in entryAt()" + e);
             }
```

```
try {
  myEntry = RCL.firstEntry();
  System.out.println("Phone number for first Entry is "
        + myEntry.getPhoneNumber());
  System.out.println("Call Type for first Entry is " +
        myEntry.getCallType());

  System.out.println(
        "Service Call Type for first Entry  is " +
        myEntry.getServiceCallType());
  System.out.println(
        "The time of the call  for first Entry is " +
        myEntry.getHour() + ":" + myEntry.getMinute());
  System.out.println(
        "The date of the call  for first Entry is " +
        myEntry.getMonth() + "-" + myEntry.getDay());
  System.out.println(
        "The duration of the call for first Entry is "
        + myEntry.getDuration());
  System.out.println(
        "The status message of the call for first Entry is "
        + myEntry.getStsMsg ());
} catch(Exception e) {
  System.out.println("Exception thrown in firstEntry()" + e);
}
try {
  myEntry = RCL.lastEntry();
  System.out.println("Phone number for last Entry is "
        + myEntry.getPhoneNumber());
  System.out.println("Call Type for last Entry is " +
        myEntry.getCallType());
  System.out.println(
        "Service Call Type for last Entry  is " +
        myEntry.getServiceCallType());
  System.out.println(
        "The time of the call  for last Entry is " +
        myEntry.getHour() + ":" + myEntry.getMinute());
  System.out.println(
        "The date of the call  for last Entry is " +
        myEntry.getMonth() + "-" + myEntry.getDay());
  System.out.println(
        "The duration of the call for last Entry  is "
        + myEntry.getDuration());
  System.out.println(
        "The status message of the call for last Entry is "
        + myEntry.getStsMsg ());
} catch(Exception e) {
  System.out.println("Exception thrown in lastEntry() " + e);
}
try {
  System.out.println("The number of incoming calls are " +
        RCL.numIncomingCalls());
} catch(Exception e) {
  System.out.println(
        "Exception thrown in numIncomingCalls() " + e);
```

```
        }
        try {
          System.out.println("The number of outgoing calls are " +
                RCL.numOutgoingCalls());
        }catch(Exception e){
          System.out.println(
                "Exception thrown in numOutgoingCalls() " + e);
        }

        try {
          System.out.println("The number of missed calls are " +
                RCL.numMissedCalls());
        } catch(Exception e) {
          System.out.println(
                "Exception thrown in numMissedCalls() " + e);
        }
        try {
          for(int i = 0; i < 5 ; i++){
              RCL.removeEntryAt(i);
          }
        } catch(Exception e) {
          System.out.println(
                "Exception thrown in removeEntryAt() " + e);
        }
        try {
          RCL.removeAll();
        } catch(Exception e) {
          System.out.println("Exception thrown in removeAll() "
                + e);
        }

    }
    else {
        System.out.println("RefreshList returned false");
    }
}
```

12. PhoneBook

12.1. Overview

The Java-based PhoneBook APIs let you access the user's phonebook data that's stored in the native database. The methods support such functionality as opening a phonebook, reading phonebook entries, creation a phonebook entry, importing a phonebook entry, removing specified phonebook entries, deleting all phonebook entries, determining available storage and so on.

12.2. Class Descriptions

The APIs for Phonebook are all located in package class com.motorola.iden.udm.

The following will be the Class Hierarchy for the UDM API:

```
java.lang.Object
  |
  +- com.motorola.iden.udm.UDM
  |
  +- com.motorola.iden.udm.PhoneBook
  |
  +- com.motorola.iden.udm.PhoneBookEntry
  |
  +-java.lang.Throwable
        |
        +-java.lang.Exception
              |
              +-com.motorola.iden.udm.UDMException
```

The following will be the Interface Hierarchy for the UDM and PhoneBook API:

```
com.motorola.iden.udm.UDMEntry
com.motorola.iden.udm.UDMList
```

12.3. Class Methods

12.3.1. UDM Method

Class for accessing the UDM databases on a device.

12.3.1.1. openPhoneBook

Returns a PhoneBook with the phone's native phonebook entries,

```
public static PhoneBook openPhoneBook(int mode) throws UDMException
```

This method returns a PhoneBook, sorted by name. `mode` must be either `READ_ONLY` or `READ_WRITE`. If you call this method and the phone's native phonebook is not ready (e.g. the SIM reads have not been completed), it throws a `UDMException`.

Calling this method is equivalent to calling `openPhoneBook(mode, NAME_SORT)`.

```
public static PhoneBook openPhoneBook(int mode, int sort)
         throws UDMException
```

This method returns a PhoneBook with the phone's phonebook entries, sorted either by name or speed dial number. `mode` must be either `READ_ONLY` or `READ_WRITE`. Sort must be either `NAME_SORT` or `SPEED_NUM_SORT`. Otherwise, this method throws an IllegalArgumentException. Note that if you sort by speed dial number, you may not be able to retrieve entries without a speed dial number. If you call this method and the phone's phonebook is not ready (e.g. the SIM reads have not been completed) it throws a `UDMException`.

The first time a MIDlet calls this method, it creates a new PhoneBook object with all the entries from the device's native phonebook. When a MIDlet calls it subsequently, it returns the same PhoneBook object, after repopulating the object with the entries from the native phonebook. Note that if your MIDlet has changed any PhoneBookEntries and hasn't committed them (with the `PhoneBookEntry.commit()`), those changes are lost.

To determine whether your application has modified a PhoneBookEntry without committing the change (with `PhoneBookEntry.commit()`) use `PhoneBookEntry.isModified()`. To determine whether the native PhoneBook database has been changed since a PhoneBook was created, use `PhoneBook.isCurrent()`.

12.3.2. PhoneBookEntry Methods

12.3.2.1. commit

Writes the data in the PhoneBookEntry to the phone's native phone book.

```
public void commit() throws UDMException
```

This method locks the native phone book, writes the data, and then unlocks the phonebook.

If this PhoneBookEntry contains only a name without a phone number, IP address, group ID, or email address, this method throws a UDMException with the string "Number Required". If this PhoneBookEntry lacks a name, this method throws a UDMException with the string "Name Required".

If the native phone database DB is busy, this method throws a UDMException with the string "Native DB is busy". This often occurs after an application calls `deleteAllPhoneBookEntries()`. When this happens, try to sleep for a period of time and try again later. It takes approximately 30 seconds to clear the phone book.

12.3.2.2. isModified

Returns true if any of this element's fields have been modified since the element was retrieved or last committed.

```
public boolean isModified ()
```

12.3.2.3. getAvailSpeedNum

Returns the next or last available speed dial number.

```
public int getAvailSpeedNum(boolean reverseOrder)
   throws UDMException
```

Use this method to generate default values for the `SPEED_NUM` field. If `reverseOrder` is false, this method returns the lowest unused speed dial number. If `reverseOrder` is true, it returns the highest unused speed dial number.

If the SIM card type is GSM SIM or ENDEAVOR SIM and `reverseOrder` is true, this method throws a UDMException with the string "PhoneBook don't support reverse order".

12.3.2.4. getFieldDataType

Returns the data type for the given field ID

```
public int getFieldDataType(int fieldID) throws UDMException
```

Use this method to find the data types for field IDs that may have different types of data in each element. This table lists the data types for the fields in a PhoneBook entry:

Field ID	Field Data Type
`TEL, SPEED_NUM, PRIV`	`UDMEntry.TYPED_STRING`
`REVISION`	`UDMEntry.DATE`
`EMAIL, FORMATTED_NAME, GRP, IP, HUB`	`UDMEntry.STRING`
`RINGER`	`UDMEntry.INT`

12.3.2.5. getInt

Returns the value of the specified integer field.

```
public int getInt(int fieldID) throws UDMException
```

If `fieldID` is not `RINGER`, this method throws a UDMException with the string "Not supported field ID". To read the available ringers, use `com.motorola.iden.call.CallReceive.playRinger(int index)`. The value for `RINGER` is an integer from 0 to the 250, which maps to one of the ringers stored on the phone. The value of the default ringer is `0xff`.

12.3.2.6. setInt

Sets the value of the specified integer field.

```
public void setInt(int fieldID, int value) throws UDMException
```

If `fieldID` is not `RINGER`, this method throws a UDMException with the string "Not supported field ID". The value for `RINGER` is an integer from 0 to the 250 that maps to one of the ringers stored on the phone. The value of the default ringer is `0xff`

12.3.2.7. getString

Returns the value of the specified string field

```
public String getString(int fieldID) throws UDMException
```

If `fieldID` is not `FORMATTED_NAME`, `GRP, HUB`, `IP` or `EMAIL`, this method throws a UDMException with the string "Not supported field ID".

12.3.2.8. setString

Sets the value of the specified string field.

```
public void setString(int fieldID, String value)
   throws UDMException
```

If `fieldID` is not `FORMATTED_NAME`, `GRP`, `IP` or `EMAIL`, this method throws a UDMException with the string "Not supported field ID".

Keep these pointers in mind when you set the value:

- The valid values for the `FORMATTED_NAME` field depend on the phone's SIM type. If SIM type is FALCON SIM and the name contains no Unicode characters, the maximum length of the name is 20 characters. If the name contains Unicode characters, the maximum length of the name is 10 characters.
- If the SIM type is any other SIM and the name contains no Unicode character, the maximum length of the name is 11 characters. If the name contains Unicode characters, the maximum length of the name is 5 characters.
- The `GRP` field can contain three or fewer digits that represent a value between 1 and 255.
- The `HUB` field can contain three or fewer digits that represent a value between 1 and 255.
- For the `IP` field, the value should be a valid IP address.
- For the `EMAIL` field, the value should be a valid email address.

12.3.2.9. getDate

Returns the value of the specified date field

```
public long getDate(int fieldID) throws UDMException
```

If `fieldID` is not `REVISION`, this method throws a UDMException with the string "Not supported field ID".

12.3.2.10. setDate

Sets the value of the specified date field

```
public void setDate(int fieldID, long value) throws UDMException
```

If `fieldID` is not `REVISION`, this method throws a UDMException with the string "Not supported field ID". The value should not be less than the date offset in milliseconds from January 1, 1970, to January 1, 1999

12.3.2.11. setTypedString

Sets the value of the specified typed string field

```
public void setTypedString(int fieldID, int typeID, String value)
   throws UDMException
```

If `fieldID` is not `TEL`, `SPEED_NUM`, or `PRIV`, this method throws a UDMException with the string "Not supported field ID". If `typeID` is not a type supported by the field, this method throws a UDMException. A list of fields and their supported types is at the PhoneBook method `getSupportedTypes()`.

- For the `TEL` field, the value should be a valid Phone Number and contain only the values in this character set `"0123456789+pwPW*#"` The maximum length of the number depends on the SIM type. If the SIM type is FALCON, the maximum length is 64 characters. Otherwise, the maximum length is 20 characters.

  "P" or "p" inserts a three-second pause into the DTMF string. "W" or "w" stops sending DTMF tones until the user presses the Send key.

- For the `PRIV` field, the value must be a valid Private Number that contains only digits. The maximum length is 18 characters.

12.3.3. PhoneBook Methods

12.3.3.1. importPhoneBookEntry

Adds the specified PhoneBookEntry to this PhoneBook

```
public PhoneBookEntry importPhoneBookEntry(PhoneBookEntry element)
   throws UDMException
```

If you opened the PhoneBook in read-only mode, this method throws a UDMException.

12.3.3.2. isSupportedField

Returns true if this PhoneBook supports the given field.

```
public boolean isSupportedField(int fieldID) throws UDMException
```

Here are the fields that this phone supports:

Fields	Supported or Not
`PhoneBookEntry.TEL`, `PhoneBookEntry.SPEED_NUM`, `PhoneBookEntry.FORMATTED_NAME`, `PhoneBookEntry.REVISION`	Supported.
`PhoneBookEntry.NAME_FAMILY`, `PhoneBookEntry.NAME_GIVEN`, `PhoneBookEntry.NAME_OTHER`, `PhoneBookEntry.NAME_PREFIX`, `PhoneBookEntry.NAME_SUFFIX`, `PhoneBookEntry.NICKNAME`	Not supported.
`PhoneBookEntry.PRIV`, `PhoneBookEntry.GRP`	If SIM Type is `SIM_GSM`, it's supported. Otherwise, it's not supported.

PhoneBookEntry.IP	If SIM Type is SIM_CONDOR or SIM_FALCON, it's supported. Otherwise, it's not supported.
PhoneBookEntry.EMAIL, PhoneBookEntry.RINGER PhoneBookEntry.HUB	If SIM Type is SIM_FALCON, they're supported. Otherwise, they're not supported.

12.3.3.3. isCurrent

Returns true if a PhoneBookEntry object has been created since the last native phonebook update.

```
public static boolean isCurrent()
```

12.3.3.4. getSupportedTypes

Returns an array of the supported types for the given field.

```
public int[] getSupportedTypes(int fieldID) throws UDMException
```

Before you call this method, call `isSupportedField(int fieldID)` to make sure the field is supported.

Here are the fields that have types and which types they support:

fieldID	Supported types
PhoneBookEntry.TEL, PhoneBookEntry.SPEED_NUM	If SIM type is FALCON, the types are PhoneBookEntry.TYPE_OTHER, PhoneBookEntry.TYPE_HOME, PhoneBookEntry.TYPE_MOBILE, PhoneBookEntry.TYPE_PAGER, PhoneBookEntry.TYPE_WORK_1, PhoneBookEntry.TYPE_WORK_2, and PhoneBookEntry.TYPE_FAX If SIM type is CONDOR, the types are PhoneBookEntry.TYPE_OTHER, PhoneBookEntry.TYPE_HOME, PhoneBookEntry.TYPE_MOBILE, PhoneBookEntry.TYPE_PAGER, PhoneBookEntry.TYPE_WORK_1, PhoneBookEntry.TYPE_FAX, and PhoneBookEntry.TYPE_MAIN. Otherwise, the only supported type is PhoneBookEntry.TYPE_MAIN;
PhoneBookEntry.PRIV	PhoneBookEntry.TYPE_PRIVATE

12.3.3.5. removePhoneBookEntry

Removes the specified PhoneBookEntry from the PhoneBook.

```
public void removePhoneBookEntry(PhoneBookEntry element)
   throws UDMException
```

If the PhoneBookEntry is not in this PhoneBook, this method throws a UDMException with "PhoneBookEntry is not in PhoneBook".

If you opened the PhoneBook in read-only mode, this method throws a UDMException with the string "PhoneBook is Read only".

If the native phone database DB is busy, this method throws a UDMException with the string "Native DB is busy". This often occurs after an application calls `deleteAllPhoneBookEntries()`. When this happens, try to sleep for a period of time and try again later. It takes approximately 30 seconds to clear the phone book.

12.3.3.6. deleteAllPhoneBookEntries

Removes all PhoneBookEntries from the list.

```
public void deleteAllPhoneBookEntries() throws UDMException
```

If you opened the PhoneBook in read-only mode, this method throws a UDMException with the string "PhoneBook is Read only"

If the native phone database DB is busy, this method throws a UDMException with the string "Native DB is busy". This often occurs after an application calls `deleteAllPhoneBookEntries()`. When this happens, try to sleep for a period of time and try again later. It takes approximately 30 seconds to clear the phone book.

12.3.3.7. getAvailableStorage

Returns an array listing the number of slots available in the native database.

```
public int[] getAvailableStorage() throws UDMException
```

The numbers returned depend on the type of SIM cared in the device. For example, a device with an Endeavor SIM returns an array of three numbers representing the number of available phone number slots, private number slots, and talkgroup slots. A device with a Condor SIM would return an array with one number representing the total number of slots available. This table shows what's returned depending on the SIM card:

SIM Type	Total Available	Phone Available	Private Available	Talkgroup Available
Endeavor SIM	N/A	100	100	30
GSM SIM	N/A	100	N/A	N/A
Condor SIM	250	N/A	N/A	N/A
Falcon SIM	600	N/A	N/A	N/A

12.4. Code Examples

The following is the code example of PhoneBook:

```
/**
 * Demo program of Motorola iDEN SDK PhoneBook APIs
 * Filename: MyPhoneBook.java
 * <p></p>
 * <hr/>
 * <b>MOTOROLA and the Stylized M Logo are registered trademarks of
 * Motorola, Inc. Reg. U.S. Pat. & Tm. Off.<br>
 * &copy; Copyright 2003 Motorola, Inc. All Rights Reserved.</b>
 * <hr/>
 *
 * @version iDEN Phonebook demo 1.0
 * @author Motorola, Inc.
 */

import com.motorola.iden.udm.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.Enumeration;
import java.lang.Thread;

public class MyPhoneBook extends MIDlet implements CommandListener {

    private Form textform;
    private Command exitCommand, checkCommand;
    private PhoneBook contacts;
    private PhoneBookEntry contact;
    private StringItem username;
    int[] type;


    /**
     * Print all contacts in a phonebook.
     * <p></p>
     * @param pbk Phonebook to be read
     */
    public void printList(PhoneBook pbk)
    {
        contacts = pbk;

        try
        {
           for (Enumeration v = contacts.elements(); v.hasMoreElements();)
        {
                /* Get one contact from phonebook */
                contact = (PhoneBookEntry)v.nextElement();
                type = contact.getFields();

                /* Get contact's name */
                username = new StringItem("name",
                    contact.getString(PhoneBookEntry.FORMATTED_NAME));
                textform.append(username);
```

```
                for (int j= 0; j<type.length; j++)
                {
                    /* Get String labels for the given field IDs */
                    System.out.print("Fields "+type[j] + " , " +
                        contact.getFieldLabel(type[j]) + " ,");

                    /* Get an integer array containing the supported type
                     * IDs for the given field ID
                     */
                    int [] alltype = contacts.getSupportedTypes(type[j]);

                    /* Get 3 types String fields from PhoneBookEntries. */
                    if (alltype.length == 0)
                    {
                        if (contact.getFieldDataType(type[j]) ==
                                UDMEntry.STRING)
                            System.out.print(contact.getString(type[j]) +
                                " ,");
                        else if (contact.getFieldDataType(type[j]) ==
                                    UDMEntry.DATE)
                            System.out.print(contact.getDate(type[j])+
                                " ,");
                        else if (contact.getFieldDataType(type[j]) ==
                                    UDMEntry.INT)
                            System.out.print(contact.getInt(type[j])+
                                " ,");
                    }
                    else
                    {   /* Get String fields with specific types
                         * in the PhoneBookEntry.
                         */
                        for (int ii =0; ii<alltype.length; ii++ )
                        {
                            System.out.print(
                                contact.getTypedString(type[j],
                                alltype[ii])+ " ,");
                        }
                    }
                    System.out.println("\n");
                }
                System.out.println("\n");
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
```

```
public MyPhoneBook() {

    textform = new Form("Hello, PhoneBook!");
    exitCommand = new Command("exit", Command.EXIT, 2);
    checkCommand = new Command("check", Command.OK, 1);
    textform.addCommand(exitCommand);
    textform.addCommand(checkCommand);
    textform.setCommandListener(this);

    int[] type;
    Enumeration v;

    String title;
    try
    {
        /* Creates a PhoneBook by read and write mode */
        contacts = UDM.openPhoneBook(UDM.READ_WRITE);
        if (contacts != null)
        {   /* Get the amount of entries (not individual numbers)
             * in the list
             */
            int no = contacts.getNumOfEntries();
            System.out.println("Number of entries is" + no);
        }

        /* Get an integer array the amount of slots available
         * on the native database.
         */
        int[] slots = contacts.getAvailableStorage();
        for (int i = 0; i < slots.length; i++)
        {
            System.out.println(slots[i]);
        }

        int index =0;
        printList(contacts);

        /* Removes a specific PhoneBookEntry from the list. */
        Enumeration e;
        e = contacts.elements();
        contact = (PhoneBookEntry)e.nextElement();
        contacts.removePhoneBookEntry(contact);

        /* Create a PhoneBookEntry for this PhoneBookEntry list. */
        contact = contacts.createPhoneBookEntry();

        contact.setString(PhoneBookEntry.FORMATTED_NAME,
            "abcdefghijklmopqrstu");
        contact.setTypedString(PhoneBookEntry.TEL,
            PhoneBookEntry.TYPE_HOME, "6795588");
```

```
                contact.setString(PhoneBookEntry.EMAIL,
                    "someone@somesite.com");
                contact.setString(PhoneBookEntry.IP, "127.0.0.1");
                contact.setInt(PhoneBookEntry.RINGER, 2);
                contact.setTypedString(PhoneBookEntry.TEL,
                    PhoneBookEntry.TYPE_WORK_1, "1234567");
                contact.commit();

                Thread.sleep(200);

                slots = contacts.getAvailableStorage();
                for (int i= 0; i < slots.length; i++)
                {
                    System.out.println(slots[i]);
                }

            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }

        public void startApp()
        {
            Display.getDisplay(this).setCurrent(textform);
        }

        public void pauseApp()
        {   }

        public void destroyApp(boolean unconditional)
        {   }

        public void commandAction(Command c, Displayable d) {
            if(c == exitCommand) {
                try {
                    contacts.close();
                }
                catch(Exception t) {   }
                notifyDestroyed();
            }
            else if (c == checkCommand)
            {
                System.out.println(PhoneBook.isCurrent());
            }
        }
}
```

12.5. Compiling & Testing PhoneBook MIDlets

- Method `PhoneBook.isCurrent()` always returns true since there is no native support for this method.
- Method `PhoneBook.getAvailableStorage()` always returns an empty array since there is no native support for this method.
- Method `PhoneBookEntry.getAvailSpeedNum()` always returns 1 since there is no native support for this method.

13. DateBook

13.1. Overview

Java-based DateBook APIs provide methods to access the user's datebook data stored within the native database. The methods support such functionality as opening the datebook, adding an except date, removing an except date, getting except dates, retrieving dates of an event, setting dates for an event, getting repeat times, setting repeat times, getting number of events in the datebook, creating datebook events, importing datebook events, getting the elements of the datebook, removing datebook events, deleting all datebook events, determining the available storage, determining the event count, and choosing a MIDlet to launch when the event times out,

13.2. Class Descriptions

APIs for DateBook are all located in package class com.motorola.iden.udm.

The following will be the class hierarchy for the UDM API:

```
java.lang.Object
|
+--com.motorola.iden.udm.UDM
|
+--com.motorola.iden.udm.DateBook
|
+--com.motorola.iden.udm.DateBookEvent
|
+--com.motorola.iden.udm.DateBookRepeatEvent
|
+--java.lang.Throwable
|
+--java.lang.Exception
         |
         +--com.motorola.iden.udm.UDMException
```

The following is the Interface Hierarchy for the UDM and DateBook API:

```
com.motorola.iden.udm.UDMEntry
com.motorola.iden.udm.UDMList
```

13.3. Method Descriptions

13.3.1. UDM Method

13.3.1.1. openDateBook

Creates a DateBook with the phone's native datebook entries.

```
public static DateBook openDateBook(int mode) throws UDMException
```

`mode` must be either `READ_ONLY` or `READ_WRITE`.

The first time a MIDlet calls this method, it creates a new DateBook object with all the entries from the device's native datebook. When a MIDlet calls it subsequently, it returns the same DateBook object, after repopulating the object with the entries from the active datebook. Note that if your MIDlet has changed any DateBookEvents and hasn't committed them (with the `DateBookEvent.commit()`), those changes are lost.

To determine whether your application has modified a DateBookEvent without committing the change (with `DateBookEvent.commit()`)., use `DateBookEvent.isModified()`. To determine whether the phone's native datebook database has been changed since the DateBook was created, use `DateBook.isCurrent()`.

13.3.2. DateBookEvent Methods

13.3.2.1. commit

Writes the data in the DateBookEvent to the phone's native datebook.

```
public void commit() throws UDMException
```

This method locks the phone's native datebook, writes the data, and then unlocks the datebook.

If this DateBookEvent is invalid, this method throws a UDMException. The DateBookEvent is invalid if its summary is null, its start time or end time is unspecified, its alarm is before current time, or it has an alarm but is an untimed event,

13.3.2.2. isModified

Returns true if any of this element's fields have been modified since the element was retrieved or last committed.

```
public boolean isModified ()
```

13.3.2.3. getFieldDataType

Returns the data type for the given field ID

```
public int getFieldDataType(int fieldID) throws UDMException
```

If `fieldID` is `START`, `END`, `ALARM`, or `REVISION`, this method returns a `UDMEntry.DATE`. If `fieldID` is `SUMMARY`, `LOCATION`, `STYLE`, `MIDLET_SUITE`, or `MIDLET`, this method returns `UDMEntry.STRING`. If `fieldID` is `RINGER`, this method returns `UDMEntry.INT`.

13.3.2.4. getDate

Returns the value of the specified date field

```
public long getDate(int fieldID) throws UDMException
```

The date is returned in milliseconds.

If you use this method with any field other than `START`, `END`, `ALARM` or `REVISION`, this method throws a UDMException with the string "Not supported field ID".

13.3.2.5. setDate

Sets the value of the specified date field.

```
public void setDate(int fieldID, long value) throws UDMException
```

If you use this method with any field other than `START`, `END`, `ALARM` or `REVISION`, this method throws a UDMException with the string "Not supported field ID".

Keep the following pointers in mind when setting these values:

- The phone's native datebook contains only events that occur between a month in the past and a year in the future. If you try to set the `START` or `END` fields to a value outside those bounds, this method throws an IllegalArgumentException with either the string "invalid start time" or "invalid end time."
- The event's `START` field must earlier than its `END` field. Otherwise the method throws IllegalArgumentException with the string "start time should be before end timer."
- The event's `ALARM` field must between 0 and 10080. (Max minutes of alarm – 7 days). Otherwise the method throws IllegalArgumentException with the string "invalid alarm value."
- The `REVISION` field is read-only. If you try to set it, this method throws a UDMException with the string "Revision is read only field".

13.3.2.6. getInt

Returns the value of the specified integer field

```
public int getInt(int fieldID) throws UDMException
```

If the `fieldID` is not `RINGER`, this method throws a UDMException with the string "Not supported field ID".

13.3.2.7. setInt

Sets the value of the specified integer field.

```
public void setInt(int fieldID, int value) throws UDMException
```

If the `fieldID` is not `RINGER`, this method throws a UDMException with the string "Not supported field ID". To read the available ringers, use `com.motorola.iden.call.CallReceive.playRinger(int index)`. The value for `RINGER` is an integer from 0 to the 250, which maps to one of the ringers stored on the phone. The value of the default ringer is `0xff`.

13.3.2.8. getString

Returns the value of the specified string field.

```
public String getString(int fieldID) throws UDMException
```

If `fieldID` is not `SUMMARY`, `LOCATION`, `STYLE`, `MIDLET_SUITE`, or `MIDLET`, this method throws a UDMException with the string "Not supported field ID".

13.3.2.9. setString

Sets the value of the specified string field.

```
public void setString(int fieldID, String value) throws UDMException
```

If `fieldID` is not `SUMMARY`, `LOCATION`, `STYLE`, `MIDLET_SUITE`, or `MIDLET`, this method throws a UDMException with the string "Not supported field ID".

Keep the following pointers in mind when setting these values:

- The `SUMMARY` and `LOCATION` fields can contain a maximum of 64 characters if the strings have no Unicode characters, or a maximum of 32 characters if the strings do have Unicode characters.
- The `MIDLET_SUITE` and `MIDLET` fields let you specify a MIDlet that is launched when this event times out. Always set the `MIDLET_SUITE` field before setting the `MIDLET` field. Note that the names are of the suite and MIDlet are case sensitive. If this method cannot find a suite or MIDlet with the specified name, this method does not set the values and throws an IllegalArgumentException.

13.3.2.10. getTypedString / setTypedString

Return or set the value of the specified typed string field.

```
public String getTypedString(int fieldID, int typeID)
   throws UDMException

public void setTypedString(int fieldID, int typeID, String value)
   throws UDMException
```

The DateBookEvent class does not contain any typed string fields. This method always throws a UDMException with the string "Not supported field ID".

13.3.3. DateBookRepeatEvent Methods

This class represents a description for a repeating pattern for a DateBookEvent element. The fields are a subset of the capabilities of the RRULE field in VEVENT defined by the vCalendar 1.0 specification from the Internet Mail Consortium (http://www.imc.org). It is used on an DateBookEvent to determine how often the Event occurs.

The following table specifies the valid values for the settable fields in DateBookRepeatEvent.

Field Ids	Set Method	Valid Values
`COUNT`	`setInt`	Any positive int
`FREQUENCY`	`setInt`	`DAILY`, `WEEKLY`, `MONTHLY`, `YEARLY`
`INTERVAL`	`setInt`	Any positive int
`END`	`setDate`	Any valid Date
`MONTH_IN_YEAR`	`setInt`	`JANUARY`, `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER`, `DECEMBER`
`DAY_IN_WEEK`	`setInt`	`SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`
`WEEK_IN_MONTH`	`setInt`	`FIRST`, `SECOND`, `THIRD`, `FOURTH`, `FIFTH`
`DAY_IN_MONTH`	`setInt`	1-31
`DAY_IN_YEAR`	`setInt`	1-366

13.3.3.1. addExceptDate

Adds a date to the repeat pattern's list of dates on which the event will not occur.

```
public void addExceptDate(long date)
```

The date should greater than date offset in milliseconds from January 1, 1970, to January 1, 1999, that means date should greater than 915148800000L. Otherwise, this method throws an IllegalArgumentException.

13.3.3.2. removeExceptDate

Removes a date from the repeat pattern's list of dates on which the event will not occur.

```
public void removeExceptDate(long date)
```

The date should greater than date offset in milliseconds from January 1, 1970, to January 1, 1999, that means date should greater than 915148800000L. Otherwise, this method throws an IllegalArgumentException.

13.3.3.3. getInt

Returns the value of the specified integer field.

```
public int getInt(int fieldID)
```

If the `fieldID` is not `COUNT`, `FREQUENCY`, `MONTH_IN_YEAR`, `WEEK_IN_MONTH`, `DAY_IN_WEEK` or `DAY_IN_MONTH`, this method throws a UDMException with the string "Not supported field ID".

13.3.3.4. setInt

Sets the value of the specified integer field.

```
public void setInt(int fieldID,  int value)
```

If the `fieldID` is not `COUNT`, `FREQUENCY`, `MONTH_IN_YEAR`, `WEEK_IN_MONTH`, `DAY_IN_WEEK` or `DAY_IN_MONTH`, this method throws a UDMException with the string "Not supported field ID".

The value of the `FREQUENCY` field must be `DAILY`, `WEEKLY`, `MONTHLY` or `YEARLY`. If not, this method throws an IllegalArgumentException with the string "value is not valid."

13.3.3.5. getDate

Returns the value of the specified date field.

```
public long getDate(int fieldID)
```

If the `fieldID` is not `END`, this method throws a UDMException with the string "Not supported field ID".

13.3.3.6. setDate

Sets the value of the specified date field.

```
public void setDate(int fieldID, long value)
```

If the `fieldID` is not `END`, this method throws a UDMException with the string "Not supported field ID".

The phone's native datebook contains only events that occur between a month in the past and a year in the future. If you try to set the `END` field to a value outside those bounds, this method throws an IllegalArgumentException

13.3.4. DateBook Methods

13.3.4.1. createDateBookEvent

Creates a DateBookEvent for this DateBook.

```
public DateBookEvent createDateBookEvent() throws UDMException
```

If there are not enough slots in the native database for a new DateBookEvent, this method throws a UDMException with the string "DateBook is full".

13.3.4.2. importDateBookEvent

Adds the DateBookEvent to this DateBook

```
public DateBookEvent importDateBookEvent(DateBookEvent element)
   throws UDMException
```

If you opened the DateBook in read-only mode, this method throws a UDMException with the string "DateBook is Read only".

13.3.4.3. isCurrent

Returns true if a DateBookEvent object has been created since the last native datebook update.

```
public static boolean isCurrent ()
```

13.3.4.4. isSupportedField

Returns true if this DateBook supports the given field.

```
public boolean isSupportedField(int fieldID) throws UDMException
```

Only these fields are supported: `DateBookEvent.START`, `DateBookEvent.END`, `DateBookEvent.ALARM`, `DateBookEvent.SUMMARY`, `DateBookEvent.LOCATION`, `DateBookEvent.REVISION`, `DateBookEvent.RINGER`, `DateBookEvent.STYLE`, `DateBookEvent.MIDLET_SUITE`, and `DateBookEvent.MIDLET`.

13.3.4.5. elements

Returns an Enumeration of DateBookEvents in this DateBook.

```
public Enumeration elements() throws UDMException
```

This method returns an Enumeration of all DateBookEvents in this DateBook. The order is not defined.

```
public Enumeration elements(long startDate, long endDate)
   throws UDMException
```

This method returns an Enumeration of all the DateBookEvents in this DateBook whose `START` field is greater than the given start date and whose `END` field is less than the given end date. The order is undefined.

If the `startDate` is greater than the `endDate`, this method throws a UDMException.

Note that the phone's native datebook contains only events that occur between a month in the past and a year in the future. If the `startDate` or `endDate` don't fall within those bounds, this method throws an IllegalArgumentException.

13.3.4.6. getEventCount

Returns an integer array of the number of used and empty slots in the native datebook database.

```
public int[] getEventCount() throws UDMException
```

Each non-repeat event occupies 1 slot. Each repeat event occupies two slots. To access the cells in the array, use the constants `NUM_OF_REPEAT_EVENTS`, `NUM_OF_NON_REPEAT_EVENTS` and `NUM_OF_EVENTS`. If the native database is closed or is no longer accessible, this method throws a UDMException.

13.3.4.7. removeDateBookEvent

Removes the specified DateBookEvent from the DateBook.

```
public void removeDateBookEvent(DateBookEvent element)
   throws UDMException
```

If the specified DateBookEvent is not in the DateBook, this method throws a UDMException.

13.4. Code Examples

The following is the code example of DateBook:

```
/**
 * Demo program of Motorola iDEN SDK DateBook APIs
 * Filename: MyDateBook.java
 * <p></p>
 * <hr/>
 * <b>MOTOROLA and the Stylized M Logo are registered trademarks of
 * Motorola, Inc. Reg. U.S. Pat. & Tm. Off.<br>
 * &copy; Copyright 2003 Motorola, Inc. All Rights Reserved.</b>
 * <hr/>
 *
 * @version iDEN Datebook demo 1.0
 * @author Motorola, Inc.
 */

import com.motorola.iden.udm.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.Enumeration;

public class MyDateBook extends MIDlet implements CommandListener
{
    private Form textform;
    private Command exitCommand, checkCommand;
    private DateBook calendars;
    private DateBookEvent dateEvent;
    private StringItem userName;
```

```
public MyDateBook()
{
    textform = new Form("Hello, DateBook!");
    exitCommand = new Command("exit", Command.EXIT, 2);
    checkCommand = new Command("check", Command.OK, 1);
    textform.addCommand(exitCommand);
    textform.addCommand(checkCommand);
    textform.setCommandListener(this);

    try
    {
        /* Create a datebook with read and write mode. */
        calendars = UDM.openDateBook(UDM.READ_WRITE);
        if (calendars != null)
        {
            /* Get number of entries in DateBook. */
            int no = calendars.getNumOfEntries();
            System.out.println(
                "Number of entries in this DateBook is " + no);
        }

        Enumeration e;

        for ( e = calendars.elements(); e.hasMoreElements(); )
        {
            dateEvent = (DateBookEvent)e.nextElement();

            int[] type;
            type = dateEvent.getFields();

            /* Get the event's detail information. */
            userName = new StringItem("subject",
                dateEvent.getString(DateBookEvent.SUMMARY));
            textform.append(userName);
            userName = new StringItem("location",
                dateEvent.getString(DateBookEvent.LOCATION));
            textform.append(userName);

            for (int j= 0; j<type.length; j++)
            {
                System.out.println("Fields " + type[j] + " " +
                    dateEvent.getFieldLabel(type[j]));
                if (dateEvent.getFieldDataType(type[j]) ==
                            UDMEntry.STRING)
                    System.out.println(dateEvent.getString(type[j]));
                if (dateEvent.getFieldDataType(type[j]) ==
                            UDMEntry.DATE)
                    System.out.println(dateEvent.getDate(type[j]));
                if (dateEvent.getFieldDataType(type[j]) ==
                            UDMEntry.INT)
                    System.out.println(dateEvent.getInt(type[j]));
            }
```

```
        /* Get how often and when this event occurs. */
        DateBookRepeatEvent rpevent = dateEvent.getRepeat();
        if (rpevent != null)
        {
            int data = rpevent.getInt(
                 DateBookRepeatEvent.FREQUENCY);
            System.out.println("FREQUENCY " +
                Integer.toString(data, 16) );

            data = rpevent.getInt(
                DateBookRepeatEvent.MONTH_IN_YEAR);
            System.out.println("MONTH_IN_YEAR " +
                Integer.toString(data, 16) );
            data = rpevent.getInt(
                DateBookRepeatEvent.WEEK_IN_MONTH);
            System.out.println("WEEK_IN_MONTH " +
                Integer.toString(data, 16) );
            data = rpevent.getInt(
                DateBookRepeatEvent.DAY_IN_WEEK);
            System.out.println("DAY_IN_WEEK " +
                Integer.toString(data, 16) );
            data = rpevent.getInt(
                DateBookRepeatEvent.DAY_IN_MONTH);
            System.out.println(data);
            System.out.println("Repeat End "+
                rpevent.getDate(DateBookRepeatEvent.END));
            long[] except = rpevent.getExceptDates();
            for (int d = 0; d < except.length; d++ )
            {
                System.out.println("Except date is " + except[d]);
            }
        }
    }

    /* Get an array of integers representing the amount of
     * additional slots that can be stored on the native database.
     * Each non-repeat event occupies 1 slot.
     * Each repeat event occupies two slots.
     */
    int[] entryField;
    entryField = calendars.getAvailableStorage();
    for (int i= 0; i < entryField.length; i++)
    {
        System.out.println(String.valueOf(entryField[i]));
    }

    entryField = calendars.getEventCount();
    for (int i= 0; i < entryField.length; i++)
    {
        System.out.println(String.valueOf(entryField[i]));
    }
```

```
            /* Create one event */
            System.out.println("Current phone time is "+
                System.currentTimeMillis());
            long currentTime = 0;
            dateEvent = calendars.createDateBookEvent();
            dateEvent.setString(DateBookEvent.SUMMARY,
                "Non-Repeat Event");
            currentTime = System.currentTimeMillis()+ 60*60000;
            dateEvent.setDate(DateBookEvent.START, currentTime);
            dateEvent.setDate(DateBookEvent.END,  currentTime + 600000);

            /* Associate a midlet_suite_name and a midlet_name
             * with this event
             */
            dateEvent.setString(MIDLET_SUITE, "midlet_suite_name");
            dateEvent.setString(MIDLET, "midlet_name");
            dateEvent.commit();

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    public void startApp()
    {
        Display.getDisplay(this).setCurrent(textform);
    }

    public void pauseApp()
    {   }

    public void destroyApp(boolean unconditional)
    {   }

    public void commandAction(Command c, Displayable d)
    {
        if(c == exitCommand)
        {
            try
            {
                calendars.close();
            }
            catch (Exception e) {    }
            notifyDestroyed();
        }
        else if (c == checkCommand)
        {
            System.out.println(DateBook.isCurrent());
        }
    }
}
```

13.5. Compiling & Testing Datebook MIDlets

- Method `DateBook.getEventCount()` always returns an empty array since there is no native support for this method.
- Method `DateBook.isCurrent()` always returns true since there is no native support for this method.
- Method `DateBook.entryIsModified(PhoneBookEntry entry)` always returns false since there is no native support for this method

14. J2ME™ ™ Networking

14.1. Overview

The i325 phone provides the following protocols specified in MIDP 2.0:

- HTTP
- HTTPS
- TCP Sockets
- SSL Secure Sockets
- Server Sockets
- UDP Sockets
- Serial Port Access

14.2. Timeouts

The timeout period for the TCP implementation on the i325 phone is 40 seconds for an open operation. The timeout period for read/write operations is about 120 seconds if the timeout flag is set to true, and about 180 seconds if the timeout flag is set to false. The lingering time for closing sockets is 10 seconds, so if a new socket is requested within this time frame and the maximum number of sockets opened has been reached, an IOException is thrown.

Applications requesting a network resource for any protocol must use one of these three methods:

```
Connector.open(String URL) – default READ_WRITE, no timeout
Connector.open(String URL, int mode) – defaults to no timeout
Connector.open(String URL, int mode, Boolean timeout)
```

The URL is the distinguishing argument that determines the difference between HTTP, UDP, Serial, and so on. The following chart details the prefixes that should be used for the supported protocols.

Table 5. Supported Protocols on the i325 Phone

Protocol	URL Format
HTTP	`http://`
HTTPS	`https://`
TCP Sockets	`socket://host:port`
SSL Secure Sockets	`ssl://host:port or ssocket://host:port`
Server Sockets	`socket://:port or serversocket://:port`
UDP Sockets	`datagram://`
Serial Port	`comm:com0 or comm:0;`

14.3. Protocols

14.3.1. HTTP

The HTTP implementation follows the MIDP 2.0 standard. The `Connector.open()` methods return an `HttpConnection` object that is then used to open streams for reading and writing. The following is a code example:

```
HttpConnection hc =
   (HttpConnection)Connector.open("http://www.motorola.com");
```

In this particular example, the standard port 80 is used, but you can specify this parameter as in the following example:

```
HttpConnection hc =
   (HttpConnection)Connector.open("http://www.motorola.com:8080");
```

The other static Connector methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection. By default, HTTP 1.1 persistency is used to increase efficiency while requesting multiple pieces of data from the same server. In order to disable persistency, set the "Connection" property of the HTTP header to "close".

14.3.2. HTTPS

The HTTPS implementation follows the MIDP 2.0 standard, except for the security aspects. The `Connector.open()` methods return an HttpsConnection object that is then used to open streams for reading and writing. The following is a code example:

```
HttpsConnection hc =
   (HttpsConnection)Connector.open("https://www.motorola.com");
```

In this particular example, the standard port 443 is used, but you can specify this parameter as in the following example:

```
HttpsConnection hc =
      HttpsConnection)Connector.open("http://www.motorola.com:8888");
```

The other static Connector methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection.

Note that only Verisign Certificates are supported in the i325 phone. The following is a list of supported features:

- SSL 3.0
- TLS 1.0
- Server Authentication

14.3.3. TCP Sockets

The low-level socket used to implement the higher-level HTTP protocol is exposed to applications via the Generic Connection Framework. The use is similar to the examples above; however, a SocketConnection is returned by the `Connection.open()` method, as in the following example:

```
SocketConnection sc =
  (SocketConnection)Connector.open("socket://www.motorola.com:8000");
```

Although similar to HTTP, notice the required port number at the end of the remote address. In the previous protocols, those ports are well known and registered so they are not required, but in the case of low level sockets, this value is not defined. The port number is a required parameter for this protocol stack.

14.3.4. SSL Secure Sockets

The low-level socket used to implement the higher-level HTTPS protocol is also exposed to applications via the Generic Connection Framework. The usage is similar to the examples above.

```
SecureSocketConnection sc =
 (SecureSocketConnection)Connector.open("ssl://www.motorola.com:8000");
```

As with non-secure sockets, the port number is a required parameter for this protocol stack.

14.3.5. Server Sockets

In addition to acting as a data requestor, some applications may act as data providers or servers. In order to accomplish this without workarounds or polling, a server socket is required. The i325 phone provides this functionality with the Generic Connection Framework. Opening a ServerSocket with the Connector object returns a ServerSocketConnection. Unlike the other networking protocols, the ServerSocketConnection does not contain any accessor methods to retrieve data, but rather only one method to accept and open a SocketConnection. This method blocks until a Socket connection is available, at which time it returns a ServerSocketConnection object. The following example illustrates this:

```
ServerSocketConnection scn =
   (ServerSocketConnection)Connector.open("socket://:8000");

ServerSocketConnection sc =
   (ServerSocketConnection)scn.acceptAndOpen();
```

The URL parameter passed in is similar to that used for TCP sockets, with the exception of the target address. In this particular instance, the target address is left blank, assuming the socket is to be opened on the local device. The port number however, is still required. The `acceptAndOpen()` method of the ServerSocketConnection object is a blocking call, so applications that utilize the particular protocol, should take this into consideration.

Note that to close the socket, you must close the associated ServerSocketConnection.

14.3.6. UDP Sockets

If networking efficiency is of greater importance than reliability, datagram (UDP) sockets are also available to the application in much the same manner as other networking protocols. The Connector object in this case returns an UDPDatagramConnection object, as is shown in the following example:

```
UDPDatagramConnection dc =
      (UDPDatagramConnection)Connector.open(
            "datagram://70.69.168.167:8000");
```

Much like low-level sockets, accessing UDP requires both a target address and a port number. The i325 phone supports a maximum outgoing and incoming payload of 1472 bytes and 2944 bytes, respectively.

14.3.7. Serial Port Access

Applications using the bottom connector (serial port) to communicate with a variety of devices are given exclusive access to the port until either the application voluntarily releases the port or the application is terminated. Much like any other networking connection, opening a serial port is not guaranteed and an exception may be thrown. If another application—native or Java—is using the port, or a cable is not attached to the device, an IOException is thrown. In the normal usage scenario, the Connector object in this instance returns a CommConnection, as is shown in the following example:

The following example shows how a CommConnection would be used to access a simple loopback program.

```
CommConnection cc = (CommConnection)
      Connector.open("comm:com0;baudrate=19200");
```

The i325 phone allows both old connection optional parameters from iDEN OEM Connection implementation and new connection parameters from MIDP 2.0. The new optional parameters are recommended as these are specified in MIDP 2.0.

These are the old parameters:

Table 6. Old Connection Optional Parameters

Parameter	Syntax	Options	Default
baudrate	baudrate = x	[300,1200,2400,4800,9600,19200,38400,57600,115200]	19200
Databits	databits = x	[8,7]	8
Stopbits	stopbits = x	1	1
parity with mapping	parity = x	[n,o,e,s,m] n=none, o = odd, e=even, s=space, m=mark	n
Flow control	flowcontrol = outflow/inflow	[n, s, h] / [n, s, h] n=none, s=software, h=hardware	N/n

Note - The following combinations of properties are not supported.
7 databits with none parity
8 databits with mark parity

8 databits with space parity
8 databits with odd parity
8 databits with even parity.

IOException will be thrown while trying to use any of the unsupported combinations in Connector.open().

And here are the new parameters:

Table 7. New Connection Optional Parameters

Parameter	Default	Description
`baudrate`	platform dependent	The speed of the port.
`bitsperchar`	`8`	The number bits per character(`7` or `8`).
`stopbits`	`1`	The number of stop bits per char(`1` or `2`)
`parity`	`None`	The parity can be `odd`, `even`, or `none`.
`blocking`	`On`	If `on`, wait for a full buffer when reading.
`autocts`	`On`	If `on`, wait for the CTS line to be on before writing.
`autorts`	On	If `on`, turn on the RTS line when the input buffer is not full. If `off`, the RTS line is always on.

14.4. Implementation Notes

As stated in the previous sections, the i325 phone supports a vast array of networking options. The networking options, however, are limited by both memory and bandwidth, which place hard restrictions on the applications. These limitations manifest themselves mainly in the number of simultaneous connections that can be opened. The following chart characterizes the boundary conditions for each networking stack.

Table 8. Concurrent Connections For The i325 Phone

Protocol	Maximum Concurrent	Notes
HTTP/ HTTPS	4	If the maximum number of HTTP Connections (4) is concurrently opened by the application and a fifth HTTP Connection is requested, connections that have not been active within the past 60 seconds are reclaimed and reused if found. Otherwise an exception is thrown.
Socket/ SecureSocket	14	If the maximum number of sockets is concurrently opened by the application, and a fifteenth socket is requested, an exception is thrown.
ServerSocket	14	It's recommend not to open more than 1 or 2 connections per MIDlet.
UDP	21	If the maximum number of sockets is concurrently opened by the application and another socket is requested, an exception is thrown to the calling application.
Serial Port	1	Only one serial port is available. If you try to open 2 concurrent serial port connections, an exception is thrown.

14.5. Tips

- Keep in mind the blocking nature of many `javax.microedition.io` and `java.io` object methods. It's recommended to spawn another thread specifically dedicated to retrieving data in order to keep the user interface interactive. If a single thread is used to retrieve data on a blocking call, the user interface becomes inactive with the end-user perceiving the application as dead.
- When the length of the data is known, reading from an InputStream using an array is faster then reading byte by byte. For example, if the content length is provided in the header of the HttpConnection, then an array of the specified size can be used to read the data.

- The InputStream and OutputStream as well as the Connection object need to be completely closed.
- An application in the suspended state can still continue to actively use the networking facilities of the i325 phone.
- The platform does not support simultaneous voice and data transmissions.

15. Location API

15.1. Overview

The i325 phone lets users and developers access GPS position information such as latitude, longitude, altitude, speed, and so on. This feature is provided as a built-in application in the phone's standard ergonomics and as a J2ME™ API developers can use to create custom AGPS-based applications. This section describes some of the phone's features that affect AGPS accuracy and availability from a J2ME™ MIDlet. This API provides access to NMEA stream of messages and can turn on the GPS chip set's NMEA capabilities.

- *Accuracy*—The i325 phone is designed to receive location fixes within a preset level of geographic accuracy as determined by the network provider. Using the Location API, J2ME™ developers can retrieve a fix; however, the location value is not guaranteed to be within this level of accuracy. The API provides methods to determine whether a given fix is accurate or not.
- Motorola strives to achieve the highest possible accuracy; however, no GPS system can provide perfect accuracy in all situations. GPS accuracy can be affected by a multitude of potential error-introducing factors, including GPS satellite signal conditions and packet data availability. Position accuracy is not guaranteed nor implied.
- *Assist Data*—AGPS uses cellular assisted data to retrieve a location fix. The Location API provides J2ME™ developers with a method to determine whether cellular assisted data is used for a given fix.

The API provides the location functionalities required for Java applications to access GPS position information such as the following:

- Latitude
- Longitude
- Altitude
- Time Stamp
- Travel Direction
- Speed
- Altitude Uncertainty
- Speed Uncertainty

The Location API uses the GPS Privacy setting in the Main Menu of the i325 phone when a MIDlet invokes the API. Based on the GPS Privacy setting value, the MIDlet does or does not have the access to the position information. The API will use the user's Privacy setting accordingly before providing position information. Some examples include:

- If the user's GPS Privacy setting is set to "Restricted", the Java API will return the position with all the attributes set to `UNAVAILABLE` and with the PositionConnection's status code set to `POSITION_RESPONSE_RESTRICTED`.
- If the user's GPS Privacy setting is set to "Unrestricted", the Java API will be able to access GPS data and will return the position.
- If the user's GPS Privacy setting is set to "By Permission", the application is suspended, as the Java API brings up a system screen to prompt the user for permission to grant position access for this application. If the user does not grant permission, the Java API will return the

position with all the attributes set to `UNAVAILABLE` and with the PositionConnection's status code set to `POSITION_RESPONSE_RESTRICTED`. After selecting one of the permission options, the user needs to resume the application.

After permission is granted, the Java API brings up a system screen to prompt the user if the Almanac data in the phone is out of date or invalid, and the phone is not provisioned for packet data service. This is done only once after the phone powers up. If the user gives permission to override Almanac data, the Java API tries to retrieve position data. If user does not grant the Almanac override, the Java API returns the position with its attributes set to `UNAVAILABLE` and the status of PositionConnection set to `POSITION_RESPONSE_NO_ALMANAC_OVERRIDE`.

15.2. Class Description

The API for the NMEA output messages is located in package com.motorola.iden.position

```
java.lang.Object
 |
 + - com.motorola.iden.position.PositionConnection
```

The PositionConnection interface supports the creation of a connection to the GPS receiver (driver). GPS position can be retrieved and status can be obtained after creating a connection. Only one connection is allowed at a time. This API must be called from a separate thread from the main application thread.

To get a PositionConnection, the MIDlet must use the generic Connector class. For example:

```
com.motorola.iden.position.PositionConnection sc =
      (com.motorola.iden.PositionConnection)Connector.open(String name);
```

String name should be one of the following:

- `name = "mposition:delay=no"`
- `name = "mposition:delay=low"`
- `name = "mposition:delay=high"`

The following descriptions of delay values are based on the default settings. These settings are carrier definable and can differ among carriers. Java has no access to change these values.

- `delay=no` This option is designed to provide the serving cell latitude and longitude to an application immediately after it requests them. Because all other attributes in the AggregatePosition class may be set to `UNAVAILABLE`, an application should use this connection only to access the serving cell latitude and longitude. This request does not make use of the GPS chipset. If the handset is outside of the network coverage area, the serving cell latitude and longitude will be set to 0.
- `delay=low` This option provides a response to the application in a few seconds. New assist data is retrieved only if no assist data exists or if the assist data is older than the Maximum Assist Data Age (MADA). This operation is transparent to the application. This option is designed to provide all the position attributes with assistance from the Location Enhanced Service (LES) Server. To exercise this option, the device needs to have packet data service. Currently the maximum response time for this type of request is 32 seconds. If the API times out, the position will be returned with appropriate status and error code. If a low-delay request is made outside of the network coverage area, then the API will not get the assist data from the LES. The fix will proceed without assist data, and the timeout will remain at the low-delay value of 32 seconds.

- `delay=high` This option provides a response to the application where delay is longer than a `delay=low` setting. It provides for an assisted or autonomous fix for the application. The phone uses existing assist data only if it is available and valid; otherwise, the location fix shall proceed autonomously. Currently, maximum response time for this type of request is 180 seconds. If the response times out, position will be returned with the appropriate status and error code.

Only one request of `getPosition()` can be made or be pending at any time. If the application makes multiple requests without getting a response to the previous request, a null position value is returned or an exception is thrown. The next section provides more detail on this method.

15.3. Method Descriptions

15.3.1. PositionConnection Methods

15.3.1.1. getPosition

Returns a position.

```
public AggregatePosition getPosition()
```

This method returns a position using the same delay setting used for `Connector.open()`.

This method is a synchronous, blocking method, which means it blocks until a response, error, or timeout occurs. Closing the PositionConnection from a separate thread can unblock these calls. Once the connection is closed, it needs to be opened again using `Connector.open()`.

If the PositionConnection is closed while a call to this method is pending or a second call has been made to this method, then this method returns a null position. Unknown errors may occur during a location fix, which may also cause null position value to be returned.

```
public AggregatePosition getPosition(String name)
```

This method returns a new position with the delay parameters specified by `name`. This method also allows an application to obtain a fix with an accurate velocity and heading direction. Note that obtaining an accurate velocity and heading direction may cause a significant delay with weak GPS signal strength. In strong GPS signal coverage this operation may take no longer than a standard fix.

The argument required for accurate velocity and heading direction is as follows:

```
String name = "delay=low;fix=extended";   // or
String name = "delay=high;fix=extended";
```

This method is a synchronous, blocking method, which means it blocks until a response, error, or timeout occurs. Closing the PositionConnection from a separate thread can unblock these calls. Once the connection is closed, it needs to be opened again using `Connector.open()`.

If the PositionConnection is closed while a call to this method is pending or a second call has been made to this method, then this method returns a null position. Unknown errors may occur during a location fix, which may also cause null position value to be returned.

15.3.1.2. requestPending

Returns true if there is a pending position request.

```
public boolean requestPending()
```

Call this method on a connection before making a new request from another thread.

15.3.1.3. getStatus

Returns the status for the last `getPosition()` call.

```
public int getStatus()
```

Call this method only after calling `getPosition()`. Use the position obtained only if `getStatus()` returns `POSITION_RESPONSE_OK`. The following is a list of the possible return values for this method:

- `POSITION_NO_RESPONSE` indicates that the device is not responding. No position information will be available, and all the attributes of the position will be set to `UNAVAILABLE`.
- `POSITION_RESPONSE_ERROR` indicates that an error occurred while retrieving the position. If possible, the cell latitude and longitude will be available, but all position's attributes will be set to `UNAVAILABLE`.
- `POSITION_RESPONSE_OK` indicates that the obtained position is a valid position. All position's attributes will be available.
- `POSITION_RESPONSE_RESTRICTED` indicates that the user has set the device so it does not provide the position information. No position information will be available, and the position's attributes will be set to `UNAVAILABLE`.
- `POSITION_WAITING_RESPONSE` indicates that the API is waiting for a response from the position device. `POSITION_WAITING_RESPONSE` will be returned if `getStatus()` method is called before `getPosition()` method.
- `POSITION_RESPONSE_NO_ALMANAC_OVERRIDE` indicates that the Almanac is outdated, and the user is restricted to override. No position information will be available, and all the attributes of the position will be set to `UNAVAILABLE`.

15.3.1.4. getNMEASentence()

Returns an NMEA Sentence for the specified type.

```
public String getNMEASentence (int type)
   throws IllegalArgumentException
```

Following are the valid NMEA message types.

- `PositionDevice.GPGGA`
- `PositionDevice.GPGLL`
- `PositionDevice.GPGSA`
- `PositionDevice.GPGSV1`
- `PositionDevice.GPGSV2`

- `PositionDevice.GPGSV3`
- `PositionDevice.GPRMC`
- `PositionDevice.GPVTG`

If the message type is other than above, this method throws an IllegalArgumentException.

If the method cannot fulfill the request for an NMEA sentence, this method returns a null string.

This first time you call this message, it turns on the GPS chip for NMEA messages. It's the application's responsibility to stop the NMEA request once it is done using it,

15.3.1.5. stopNMEASentence

Stops the NMEA request and turns off the GPS chip after 10 seconds.

```
public void stopNMEASentence()
```

This method stops only the NMEA access and keeps the connection open so the application can use the connection to retrieve the position fix or reuse it for NMEA messages.

15.3.2. AggregatePosition Methods

15.3.2.1. getResponseCode

Returns the response code for this position.

```
public int getResponseCode ()
```

The following is a list of returned response codes:

- `POSITION_OK` indicates that the obtained position is valid and accurate.
- `ACC_NOT_ATTAIN_ASSIST_DATA_UNAV` indicates that the location fix has timed out. The fix could not be accurately obtained since assistance data was not unavailable.
- `ALMANAC_OUT_OF_DATE` indicates that the Almanac is out of date.
- `ACCURACY_NOT_ATTAINABLE` indicates that the location fix has timed out, and the requested accuracy is not attainable.
- `BATTERY_TOO_LOW` indicates that the battery is too weak to retrieve a fix.
- `FIX_NOT_ATTAIN_ASSIST_DATA_UNAV` indicates that the location fix has timed out because a fix is not attainable, and assist data is unavailable.
- `FIX_NOT_ATTAINABLE` indicates that the location fix has timed out because a fix is not attainable.
- `GPS_CHIPSET_MALFUNCTION` indicates that the GPS chipset is malfunctioning.
- `UNAVAILABLE` indicates that an unknown error has occurred. This is the default response code.

These response codes are used in conjunction with `PositionConnection.getStatus()` to determine the quality of the retrieved position. These values are valid only when either `POSITION_RESPONSE_ERROR` or `POSITION_RESPONSE_OK` have been returned.

The following table shows the possible combinations of response codes for these two methods:

PositionConnection Status Values	Response Codes
POSITION_RESPONSE_OK	POSITION_OK ACCURACY_NOT_ATTAINABLE ACC_NOT_ATTAIN_ASSIST_DATA_UNAV
POSITION_RESPONSE_ERROR	FIX_NOT_ATTAINABLE FIX_NOT_ATTAIN_ASSIST_DATA_UNAV BATTERY_TOO_LOW GPS_CHIPSET_MALFUNCTION ALMANAC_OUT_OF_DATE, UNAVAILABLE

15.3.2.2. getAssistanceUsed

Checks if a fix has been retrieved using assistance.

```
public boolean getAssistanceUsed ()
```

15.4. Code Examples

```
void getViaPositionConnection() throws IOException {
  PositionConnection c = null;
  String name = "mposition:delay=low";
  try{
    c = (PositionConnection)Connector.open(name);
    AggregatePosition oap = c.getPosition();
    // Returns the AggregatePosition which contains the position
    // using the parameter passed when connection was opened.
    // Application should only check status by calling getStatus()
    // after getPosition() or getPosition(String name) returns.
    // Otherwise, it returns the same status and is
    // considered an invalid call of getStatus().
    // check the status code for permission and almanac over ride
    if(c.getStatus() ==
            PositionConnection.POSITION_RESPONSE_RESTRICTED)
    {
      //  means user has restricted permission to get position
    }
    else if(c.getStatus() ==
            PositionConnection.POSITION_RESPONSE_NO_ALMANAC_OVERRIDE)
    {
      // means device has Almanac out of date and
      //the user has not granted to override
    }
    else if(c.getStatus() ==
            PositionConnection. POSITION_NO_RESPONSE)
    {
      //  means no response from device
    }
```

```
if (oap != null ) {
  if(c.getStatus() ==
         PositionConnection.POSITION_RESPONSE_OK)
  {
    // Good position
    // Check for any error from device on position
    // Application needs to check for null position
    if(oap.getResponseCode() == PositionDevice.POSITION_OK) {
      // no error in the position
      if(oap.hasLatLon()) {
        // int value of Latitude and Longitude of the position in
        // arc minutes multiplied by 100,000 to maintain accuracy
        // or UNAVAILABLE if not available
        int lat = oap.getLatitude();
        int lon = oap.getLongitude();
        // String representation of the Latitude and Longitude.
        String LATDEGREES = oap.getLatitude(Position2D.DEGREES);
        String LONGDEGREES = oap.getLongitude(Position2D.DEGREES);
      }
      if(oap.hasSpeedUncertainty()) {
        // speed and heading value are valid
        int speed = oap.getSpeed();
        if (hasTravelDirection()) {
            // heading is available
               int travelDirection = oap.getTravelDirection();
        }
      }
      if(oap.hasAltitudeUncertainty()) {
        int alt = oap.getAltitude(); //altitude of position
                                     // in meters.
       }
    }
    // handle the errors…or request again for good position
    // or display message to the user.
    else if(oap.getResponseCode() ==
       PositionDevice.ACCURACY_NOT_ATTAINABLE) {
      // the position information was provided but enough
      // accuracy may not be attainable
    }
    else if(oap.getResponseCode() ==
      PositionDevice.ACC_NOT_ATTAIN_ASSIST_DATA_UNAV) {
      // the position information was provided but enough
      // accuracy, assistant data unavailable
    }
  } // end of position response ok
  else if(c.getStatus() ==
           PositionConnection.POSITION_RESPONSE_ERROR)
  {
    //  indicate an error occurred while getting the position
    if(oap.getResponseCode() ==
           PositionDevice.FIX_NOT_ATTAINABLE) {
      // means position information not provided (timeout)
    }
```

```
        else if(oap.getResponseCode() ==
               PositionDevice.FIX_NOT_ATTAIN_ASSIST_DATA_UNAV) {
          // means position information not provided (timeout) and
          // assistant data unavailable
        }
        else if(oap.getResponseCode() ==
               PositionDevice.BATTERY_TOO_LOW) {
          // means battery is too low to provide fix
        }
        else if(oap.getResponseCode() ==
                PositionDevice.GPS_CHIPSET_MALFUNCTION) {
          // means GPS chipset malfunction
        }
        else if(oap.getResponseCode() ==
                PositionDevice.ALMANAC_OUT_OF_DATE) {
          // means almanac out of date to get fix
          // This scenario occurs when user overrides almanac but
          // device is not packet data provisioned
        }
        else {
          // Unknown error occurs
        }
      }// end of position response error
       // position is null
} finally {
   if ( c != null)
   c.close();
}
```

New positions can be obtained using the following method on the same `PositionConnection` object until the `close()` method is called.

```
AggregatePosition cell = c.getPosition("delay=no");
```

Or

```
AggregatePosition oap = c.getPosition("delay=low");
```

Or

```
AggregatePosition oap = c.getPosition("delay=high");
```

In addition, to obtain better accurate speed and direction

```
AggregatePosition oap = c.getPosition("delay=low;fix=extended");
```

Or

```
AggregatePosition oap = c.getPosition("delay=high;fix=extended");
```

The following is an NMEA code example:

```
try
{
    PositionConnection posCon =
       (PositionConnection)Connector.open("mposition:delay=low");

   String temp1 = posCon.getNMEASentence(PositionDevice.GPGGA);
   if(posCon.getStatus() == POSITION_RESPONSE_OK)
   {
       if(temp1 != null && temp1.equals(""))
       {
          // valid GPGGA string, parse it to extract
          // the required information
       }
      else if(posCon. getStatus() == POSITION_RESPONSE_RESTRICTED)
      {
         // User has not granted permission to access
         // its location information
      }
      else if (posCon. getStatus() ==
                POSITION_RESPONSE_NO_ALMANAC_OVERRIDE)
     {
          // User has not granted permission to override
          // its almanac information
      }
     else
      {
            // unusual error occurred
      }
}
catch(IllegalArgumentException ie) {
}
catch(Exception ex) {
}
```

15.5. Tips

- The GPS receiver requires access to both the iDEN network and GPS satellite signals to obtain rapid fixes. It is recommended that once the first fix is obtained, the application monitor the response codes and vary the times between position requests accordingly. This recommendation is to handle the real world case where an application requests fixes rapidly (less than 10 seconds apart) and then loses network and GPS coverage (by entering a parking structure, basement, etc.) The GPS system will continue to try to find the unit's position and will go into a longer integration or acquisition mode that, once started, may take so long to finish that it may miss GPS signals once back in coverage. The recommended practice is to make fixes rapidly until a response code of `FIX_NOT_ATTAINABLE` or `FIX_NOT_ATTAIN_ASSIST_DATA_UNAV` is returned several times in a row (for about 10 requests for `delay=low` and about 5 requests for `delay = high`). After this occurs, the application may wish to start the acquisition over from the beginning in anticipation that the phone might be back in GPS coverage. To do so, the application must wait 15 to 20 seconds after receiving the last response code before

requesting a new fix. After this pause, the application can continue requesting fixes at its normal frequency.

- GPS subsystem requires about one second to calculate a new fix, so any request for a new fix during this one-second period may result in the exact same position information including the time stamp. Therefore it is recommended that an application request a new position no more than once per second.
- If an application needs continuous position, use "`delay=low`" once and "`delay=high`" thereafter even if the first fix does not succeed. The reason for this is because of network failures. When there is a network failure, there is a 12 to 24 second communication timeout from the LES.
- Use "`delay=no`" if the application needs only the cell latitude and longitude. This does not use AGPS chip on device.
- Applications must handle all response codes returned by the AggregatePosition `getResponseCode()` method and the PositionConnection `getStatus()` method. `getStatus()` provides the connection's status after the fix and user interaction status with regards to permission. `getResponseCode()` provides information about the position itself.
- Applications must always check the speed uncertainty value before using speed and heading. Although it is counter-intuitive, the presence of speed uncertainty denotes that the speed and heading value are accurate. Therefore, if a call to `hasSpeedUncertainty()` returns true, the speed and heading returned by the API are valid.
- If an application calls `getPosition(String name)` method with the "`fix=extended`" tag, this method will return accurate velocity and heading direction; however, there is a time penalty since it takes longer to calculate the accurate velocity and heading direction when the method is called.
- The method `PositionConnetion.getStatus()` provides the status of the connection when the method `PositionConnection.getPosition()` was called. Whereas, `AggregatePosition.getResponseCode()` returns the detailed response code
- Getting a position for the first time after the phone powers on is referred as a "cold start". A position retrieved within ten seconds of the previous fix is referred to as a "hot start". A position retrieved after ten seconds of the previous fix is a "warm start". After 1 hour since the last fix will set the device back to "cold start". Therefore, "hot start" is the quickest way of retrieving a fix.
- For i325 it is highly recommended that antenna remain extended all the time while getting fixes.
- There is a battery impact when the NMEA API is used heavily.
- If the application will need NMEA data again in less than 10 seconds, there is no value in calling `stopNMEASentence`() because the GPS chip will stay on for 10 seconds after calling `stopNMEASentence`().
- First call of `getNMEASentence`() will turn on the GPS chip and it stays on until application calls `stopNMEASentence()`.

16. Crypto APIs

16.1. Overview

To complement SSL/TLS/HTTPS and enrich secure Java applications, the i325 phone includes a set of lightweight cryptography APIs that provide flexible and customizable end-to-end application-layer security in the J2ME™ environment. A rich variety of cryptographic mechanisms and algorithms are incorporated into these APIs, thus providing confidentiality, integrity and authentication. Cryptographic algorithms and schemes supported include: message digest (MD5 and SHA-1), secure random number generator (FIPS186 RNG), ciphers (DES, DESede, AES, RC4, and others), digital signatures (ECDSA and others) and key agreement (DH and ECDH).

16.2. Class Descriptions

The Crypto APIs are located in the packages com.motorola.iden.crypto and com.motorola.iden.security.

```
java.lang.Object
   |
   +-com.motorola.iden.crypto.Cipher
   |
   +-com.motorola.iden.crypto.KeyAgreement
   |
   +-com.motorola.iden.security.MessageDigestSpi
   |
   +-com.motorola.iden.security.MessageDigest
   |
   +-com.motorola.iden.security.Signature
```

16.2.1. MessageDigest Description

MessageDigest is a one-way hash function that takes arbitrary-sized data and outputs a fixed-length hash value. All the information in the message is used to construct the message digest, but the message cannot be recovered from the hash. The message digest provides data integrity.

Algorithms MD5 and SHA-1 are supported in this platform.

The MessageDigest class provides applications with the functionality of a message digest algorithm, such as MD5 or SHA-1.

- SHA-1 is a basic hash function that takes an entire message (or several parts of a single message submitted in separate blocks) and produces a 160-bit message digest value.
- MD5 is a hash function that takes an entire message (or several parts of a single message submitted in separate blocks) and produces a 128-bit message digest value.

16.2.2. Cipher Description

Encryption is a tool used to protect data. Typical uses are to protect files in a file system or to encrypt network communications.

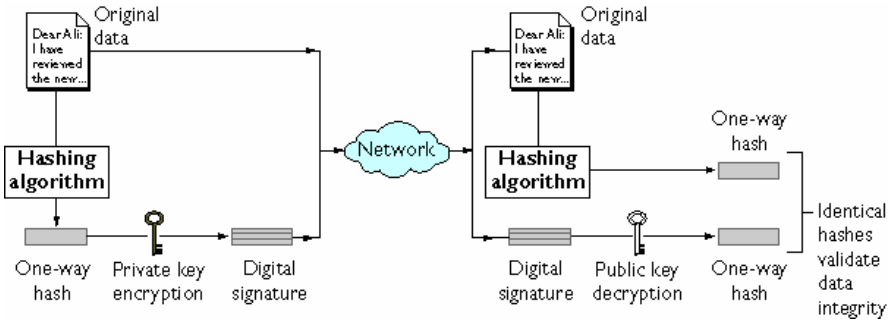
The i325 phone supports two kinds of ciphers:

- Symmetric Ciphers use a single secret key to encrypt and decrypt data.
- Asymmetric Ciphers use a pair of keys. One key is public and may be freely distributed. The other key is private and should be kept secret. Data encrypted with either key can be decrypted using the other key.

This class provides the functionality of a cryptographic cipher for encryption and decryption

16.2.3. Signature Description

A digital signature is simply a message digest that has been processed with a signer's private key. The signature can be passed around with the data, providing proof that whoever signed the data had access to the private key.



The Signature class provides the functionality of signing and verifying a digital signature.

16.2.4. KeyAgreement Description

KeyAgreement can establish shared secrets without exchanging a secret key. KeyAgreement relies on public-public key pairs, just like asymmetric encryption. Your own private key and another party's public key generate the shared secret. The generated shared secret can be used as a key for symmetric encryption.

The class KeyAgreement that is located in package com.motorola.iden.crypto contains the method of generating and verifying a digital signature. Algorithms Diffie-Hellman (DH) and ECC Diffie-Hellman (ECDH) are supported. For algorithm DH, standard ANSI X9.63 KDF is followed and for ECDH, ANSI X9.42 KDF is followed. For ECDH, only curve WTLS-7 (160 bits) is supported.

16.3. Method Descriptions

16.3.1. MessageDigest Methods

16.3.1.1. getInstance

Creates a MessageDigest instance.

```
public static MessageDigest getInstance(String algorithm)
   throws NoSuchAlgorithmException
```

This method generates a MessageDigest instance, which implements one of the above algorithms.

`algorithm` is the name of the algorithm requested; for example, "MD5" or "SHA".

16.3.1.2. update

Updates the digest with the specified bytes.

```
public void update(byte[] input, int offset, int len)
```

This method updates the message digest with an array of bytes that represents one of several parts of a single message. A message can be submitted in separate blocks. This method can be called multiple times.

`input` is the array of bytes. `offset` is the offset to start from in the array of bytes. `len` is the number of bytes to use.

16.3.1.3. digest

Returns a completed digest, created from the parts specified with calls to `update()`.

```
public byte[] digest()
```

After you finish updating the entire message, this method finishes the operation and produces the digest.

16.3.2. Cipher Methods

16.3.2.1. getInstance

Creates a Cipher instance.

```
public static final Cipher getInstance(String transformation)
   throws NoSuchAlgorithmException, NoSuchPaddingException
```

This method generates a Cipher instance that represents a certain cipher algorithm and possible associated padding scheme.

`transformation` is the name of the transformation in the form "algorithm/mode/padding" or "algorithm"; for example, `"DES/CBC/PKCS5Padding"` or `"DES"`.

If the transformation is specified by algorithm only, the mode and padding are set to the default values for the algorithm provider.

The following table lists all supported cipher algorithms, modes, and padding.

Table 9. Supported Cipher Algorithms on the i325 Phone

Algorithm	Mode	Padding
DES	ECB;CBC;CFB;OFB	PKCS5Padding
DESede	ECB;CBC;CFB;OFB	PKCS5Padding
AES	ECB;CBC;CFB_128;OFB_128	PKCS5Padding
ARC4 (or RC4)	-----	-----

16.3.2.2. init

Initializes a Cipher instance with an operation mode, key, and algorithm specifications.

```
public final void init(int opmode, Key key,
   AlgorithmParameterSpec params)
   throws InvalidKeyException, InvalidAlgorithmParameterException
```

Before you perform any other operation on the Cipher, call this method to initialize it with the operation mode (encrypt or decrypt), a key, and the proper algorithm parameters (such as the initial vector).

`opmode` is the operation mode of this cipher (e.g. `ENCRYPT_MODE`, `DECRYPT_MODE`). `key` is the encryption key. `params` is the algorithm parameters.

16.3.2.3. update

Updates the cipher with the specified bytes.

```
public final byte[] update(byte[] input, int offset, int len)
   throws IllegalStateException
```

This method places information into the cipher to start or to continue a multiple-part encryption or decryption operation..

`input` is the input buffer. `offset` is the offset in `input` where the input starts. `len` is the input length

16.3.2.4. doFinal

Returns a completed cipher, created from the parts specified with calls to `update()`.

```
public final byte[] doFinal()
   throws IllegalStateException, IllegalBlockSizeException,
   BadPaddingException
```

This method finishes a multiple-part encryption or decryption operation and produces the cipher (if the operation was encryption) or plain text (if the operation was decryption).

16.3.3. Signature Methods

16.3.3.1. getInstance

Creates a Signature instance.

```
public static Signature getInstance(String algorithm)
   throws NoSuchAlgorithmException
```

This method creates a Signature instance that implements the specified signature.

`algorithm` is the name of the algorithm, such as `"ECDSA"`.

16.3.3.2. initSign

Initializes the Signature for signing with the specified key.

```
public final void initSign(PrivateKey privateKey)
   throws InvalidKeyException
```

Before you perform any signing operation, you must call this method to specify the private key.

`privateKey` is the private key of the identity whose signature is to be generated.

16.3.3.3. initVerify

Initializes the Signature for verification with the specified key.

```
public final void initVerify(PublicKey publicKey)
   throws InvalidKeyException
```

Before you perform any verification operation, you must call this method to specify the public key.

`publicKey` is the public key of the identity whose signature is going to be verified.

16.3.3.4. update

Updates the data to be signed or verified with the specified bytes.

```
public final void update(byte[] data, int offset, int len)
   throws SignatureException
```

This method updates the data to be signed or verified with the specified array of bytes.

`data` is the array of bytes. `offset` is the offset to start from in the array of bytes. `len` is the number of bytes to use, starting at `offset`.

16.3.3.5. sign

Returns the signature for the data specified with `update()`.

```
public final byte[] sign() throws SignatureException
```

This method returns the signature bytes of the input data. The format of the signature depends on the underlying signature scheme. Calling this method resets this signature object to the state it was in when initialized with `initSign()`.

16.3.3.6. verify

Returns true if the specified signature matches the data specified with `update()`.

```
public final boolean verify(byte[] signature)
   throws SignatureException
```

This method verifies that the specified signature is for the data specified with `update()`.

Calling this method resets this signature object to the state it was in when initialized with `initVerify()`.

16.3.4. KeyAgreement Methods

16.3.4.1. getInstance

Creates a KeyAgreement instance.

```
public static KeyAgreement getInstance(String algorithm)
   throws NoSuchAlgorithmException
```

This method generates a `KeyAgreement` object that implements the specified key agreement algorithm. In the current implementation, algorithm DH and ECDH are available.

algorithm is the name of the key agreement algorithm; that is “DH” or “ECDH”.

16.3.4.2. init

Initializes the KeyAgreement.

```
public final void init(Key key, AlgorithmParameterSpec params)
   throws InvalidKeyException, InvalidAlgorithmParameterException
```

Before you can use the KeyAgreement, you must call this method to initialize it with the given key and set of algorithm parameters.

`key` is the party's private information. For example, in the case of the Diffie-Hellman key agreement, this would be the party's own Diffie-Hellman private key.

`params` is the key agreement parameters.

16.3.4.3. doPhase

Updates the KeyAgreement with a key received from one of the other parties involved in this key agreement.

```
public final Key doPhase(Key key, boolean lastPhase)
   throws InvalidKeyException
```

`key` is the key for this phase. For example, in the case of Diffie-Hellman between two parties, this would be the other party's Diffie-Hellman public key.

`lastPhase` is a Boolean flag that indicates whether this is the last phase of this key agreement. Currently, only one phase is supported so this argument should always be true. Using false causes this method to throw an exception.

16.3.4.4. generateSecret

Returns the shared secret based on the keys obtained from `init()` and `doPhase()`.

```
public final byte[] generateSecret()
```

This method resets this KeyAgreement instance, so that it can be reused for further key agreements. Unless this key agreement is reinitialized with one of the `init()` methods, the same private information and algorithm parameters are used for subsequent key agreements.

16.4. Example Code

16.4.1. MessageDigest Example #1

```
public CDemo1()
{
      byte[] message1 = new byte[25];
      byte[] message2 = new byte[250];
      byte[] digest;
      try{
            //get an Instance of MessageDigest whose algorithm
            //is MD5
            MessageDigest md = MessageDigest.getInstance ("MD5");

            //update message1 into MessageDigest context
            md.update(message1,0,25);

            //update part of message2 (start at element 2, length //125)
            // into MessageDigest context
            md.update(message2,2,125);

            //finalize and get MessageDigest
            digest = md.digest();
      } catch (NoSuchAlgorithmException) {}
}
```

16.4.2. MessageDigest Example #2

```
public CDemo2()
{
      byte[] message1 = new byte[25];
      byte[] message2 = new byte[250];
      byte[] digest;
      try{
            //get an Instance of MessageDigest whose algorithm
            //is SHA-1
            MessageDigest sha = MessageDigest.getInstance("SHA");

            //update message1 into MessageDigest context
            sha.update(message1,0,25);

            //update part of message2 (start at element 2, length //125)
            // into MessageDigest context
            sha.update(message2,2,125);

            //finalize and get MessageDigest
            digest = sha.digest();
      } catch (NoSuchAlgorithmException) {}
}
```

16.4.3. Cipher Example

```
public cipherdemo1()
{
      //message needs to be encrypted
      String info = "Hello World!"

      //cipher
      byte[] cipher;

      //Decrypted message
      String output;

      try {
            //get a cipher instance for encryption
            Cipher A = Cipher.getInstance("DES/CBC/PKCS5Padding");

            //get a cipher instance for decryption
            Cipher B = Cipher.getInstance("DES/CBC/PKCS5Padding");

            //setup a des key
            byte key_input[] = {0,1,2,3,4,5,6,7};

            //key instance
            //DES_Key implements interface Key
            DES_Key key = new DES_Key(key_input);

            //initial vector
            byte [] iv;
```

```
            //init cipher A
            A.init(Cipher.ENCRYPT_MODE, key);

            //get generated IV
            iv = A.getIV();

            //encrypt
            cipher = A.doFinal(info.getBytes());

            //new IvParameterSpec for decryption
            IvParameterSpec ips = new IvParameterSpec(iv);

            //init cipher for decryption
            B.init(Cipher.DECRYPT_MODE,key,(AlgorithmParameterSpec)ips);

            //decrypt the message
            byte out[] = B.doFinal(encrypted3);

            //get the decrypted info
            output = new String(out, 0, out.length);
      }
      catch (Exception e) {
      }
}
```

16.4.4. Signature Example

```
public CDemo3()
{
      Signature sig, verify;

      try {
            //get new Signature instance for signing.
            sig = Signature.getInstance("ECDSA");

            //setup ECDSAParameterSpec for initialization
            ECDSAParameterSpec ecdsaparameter = new
                  ECDSAParameterSpec(Security.WTLS7,null);
            sig.setParameter(ecdsaparameter);

            //initialize for signing
            sig.initSign((ECC_PrivateKey)privatekey);

            //update the message to be signed
            sig.update("testtesttest".getBytes(),0,12);

            //get the signature (s-value)
            byte[] signature = sig.sign();

            //get the r-value and store it into ecdsaparameter
            ecdsaparameter = (ECDSAParameterSpec)sig.getParameter();

            //get new Signature instance for verifying
            verify = Signature.getInstance("ECDSA");
```

```
            //set ECDSAParameterSpec for verifying
            //setup both curve and r-value
            verify.setParameter(ecdsaparameter);

            //initialize for verifying
            verify.initVerify((ECC_PublicKey)publickey);

            //update the message to be verified
            verify.update("testtesttest".getBytes(),0,12);

            //verify
            boolean b = sig2.verify(signature);

      }
      catch (Exception e){
      }
}
```

16.4.5. Key Address Example

```
public CDemo4()
{
      //initialize variables used in this Key Agreement
      KeyAgreement dh;
      KeyAgreement dh2;

      KeyPair keypair;
      KeyPair keypair2;

      DHParameterSpec dhspec;
      DHParameterSpec dhspec2;

      KeyPairGenerator dhgen;
      KeyPairGenerator dhgen2;

      PublicKey publickey;
      PublicKey publickey2;

      PrivateKey privatekey;
      PrivateKey privatekey2;

      byte[] BobS;
      byte[] AliceS;

      int i;
      try {
            //BOB
            //create dhspec
            dhspec = new DHParameterSpec(p,g,q);

            //create dhgen
            dhgen = KeyPairGenerator.getInstance("DH");

            //init dhgen
            dhgen.initialize(dhspec);
```

```
            //gen keypair
            keypair = dhgen.generateKeyPair();

            //get publickey and privatekey for dh
            publickey = keypair.getPublic();
            privatekey = keypair.getPrivate();

            //Alice
            //create dhspec
            dhspec2 = new DHParameterSpec(p,g,q);

            //create dhgen
            dhgen2 = KeyPairGenerator.getInstance("DH");

            //init dhgen
            dhgen2.initialize(dhspec2);

            //gen keypair
            keypair2 = dhgen2.generateKeyPair();

            //get publickey and privatekey for dh
            publickey2 = keypair2.getPublic();
            privatekey2 = keypair2.getPrivate();

            //get dh
            dh = KeyAgreement.getInstance("DH");

            //init dh
            dh.init((DH_PrivateKey)privatekey,dhspec);

            //doPhase
            dh.doPhase((DH_PublicKey)publickey2,true);

            //generate secret key using Bob's private key and Alice's
            //public Key
            BobS = dh.generateSecret();


            //get dh
            dh2 = KeyAgreement.getInstance("DH");

            //init dh
            dh2.init((DH_PrivateKey)privatekey2,dhspec2);

            //doPhase
            dh2.doPhase((DH_PublicKey)publickey,true);

            //generate secret key using Alice's private key and Bob's
            //public Key
            AliceS = dh2.generateSecret();

        } catch (Exception e) {
        }
    }
```

16.5. Tips

- In order to use DES, DESede, AES, and ARC4, a MIDlet must implement the Key interface.
- DES supports 56-bit key (8 bytes, including parity). DES key parity check and weak key detection are not supported.
- DESede, also called 3DES ("triple DES"), supports 168-bit keys (24 bytes, including parity). Parity check and weak key detection are not supported.
- AES supports 128, 192 or 256-byte key.
- ARC4, also called RC4, supports a key size that is less than 256 bits.

16.6. Compiling & Testing Cryptography Enhanced MIDlets

These remarks are only applicable to the stub classes for emulators.

- Instead of executing actual cryptographic operations, console messages are displayed for certain operations. This allows rudimentary debugging of applications without actual cryptographic operations.

17. Look and Feel (LnF)

17.1. Overview

The main purpose of the LnF API is to provide facilities to modify the graphical user interface and interface related behavior of J2ME™ LCDUI components without breaking backward compatibility and without modifying the standardized J2ME™ LCDUI APIs. The LnF allows developers to modify how the standard user interface (UI) components available in J2ME™ (javax.microedition.lcdui) look and respond to certain user interactions without modifying the base code of every component. The LnF also provides an API to allow developers to plug-in different LnFs (styles) without compromising the standardized API provided by MIDP 1.0+. The LnF allows developers to modify font settings, color settings, border settings, and icon sets of UI components.

The LnF API allows specifying the style used by a particular family of UI components. Style refers to the border that surrounds the component, the color scheme and the font settings used by the component; in other words, the look-and-feel. A family of UI components refers to all components of the same Java class, including any that may have been instantiated already.

In addition to styles, LnF 2.0 provides a much more flexible and richer functionality: it allows developers to modify the geometry of the UI components by allowing them to overwrite the paint functionality used by the UI components. That means that the developer is now capable of implementing the low-level graphics code (refer to MIDP Canvas and Graphics) that will be invoked by the component when it requires the rendering of its contents. In order to achieve this capability, developer must comply fully with the framework. That means that developer must correctly overwrite all methods required by the LnF framework.

The main two functionalities that a developer must overwrite are the following: the low-level paint functionality and the preferred dimension request functionality. The LCDUI/LnF framework will first query the overwritten version of the LnF what the preferred dimensions of a particular component will be. Every UI component should have the appropriate rectangular space to render fully its own contents. In order for developers to calculate these preferred dimensions accurately, the framework passes references to a component's content. Once the preferred dimensions are obtained, the framework will proceed with the layout calculation. The layout calculation (refer to MIDP 2.0 Form/Item section and iDEN MIDP 2.0 layout section) will use the preferred dimensions to allocate the correct space for every UI component and determine its location within the screen space. Once the calculation is completed, the paint cycle of the Screen will invoke the overwritten paint method of every component while passing all the applicable information regarding the state and contents of the component in turn.

The LCDUI/LnF framework is a composition of UI components and their corresponding LnF classes. The UI component classes (i.e. any Item subclass such as ImageItem, Gauge, etc.) communicate through a defined API with their corresponding LnF class. The API uses 2 basic parameters: an array of Objects (`Object[]`) to pass in the contents of the Item (i.e. the Image and label of an ImageItem) and a single integer to specify the mode. The mode is just an arrangement of bits, which specify the state of the component such as highlighted (focused), and its mode (radio-button, checkbox, etc.).

Depending on the nature of the Item, the array of Objects will contain different Objects. The implementations of LnF classes must typecast every Object within the array to its appropriate class in order to use it. Also, depending on the nature of the Item, the mode will have to be "decoded" using the appropriate masks to determine the state: highlighted, selected, etc.

17.2. Class Description

The API for the LnF is located in package com.motorola.iden.lnf.

```
com.motorola.iden.lnf.LookAndFeel
|
+- com.motorola.iden.lnf.DisplayableLookAndFeel
|  |
|  +- com.motorola.iden.lnf.SystemLookAndFeel
|
+- com.motorola.iden.lnf.WidgetLookAndFeel
   |
   +- com.motorola.iden.lnf.ItemLookAndFeel
   |  |
   |  +- com.motorola.iden.lnf.CheckboxItemLookAndFeel
   |  +- com.motorola.iden.lnf.ChoiceGroupItemLookAndFeel
   |  +- com.motorola.iden.lnf.DateFieldItemLookAndFeel
   |  +- com.motorola.iden.lnf.GaugeItemLookAndFeel
   |  +- com.motorola.iden.lnf.ImageItemLookAndFeel
   |  +- com.motorola.iden.lnf.StringItemLookAndFeel
   |  +- com.motorola.iden.lnf.TextFieldItemLookAndFeel
   |
   +- com.motorola.iden.lnf.CommandAreaLookAndFeel
   +- com.motorola.iden.lnf.ScrollbarLookAndFeel
   +- com.motorola.iden.lnf.TickerLookAndFeel
   +- com.motorola.iden.lnf.TitleLookAndFeel

com.motorola.iden.lnf.LookAndFeelEngine
```

The most important class is the LookAndFeelEngine. The LookAndFeelEngine allows developers to control the LnF by allowing developers to replace existing LookAndFeel implementations for customized ones. The LookAndFeelEngine also provides an API to get references to the existent LnF classes so its inner attributes such as Border, Font, ColorPalette, and JustificationStyle can be modified as well.

17.3. Code Examples

The following is a set of examples to modify the LnF. These examples are organized as follows:

1. Replacing an existing LookAndFeel with a different customized version
2. Creating a new CheckboxItemLookAndFeel

Example 1 describes how to replace an existing LookAndFeel with a different one. Developers must create a valid LnF class that extends the original one used by the framework and overwrite the methods accordingly to his/her needs.

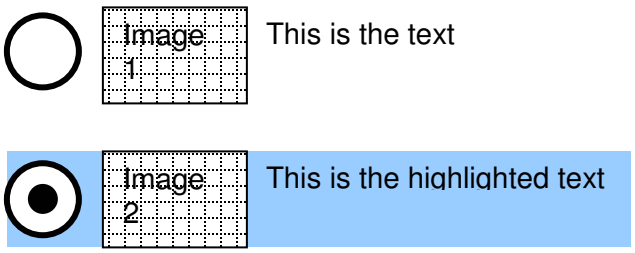
Example 2 illustrates the creation of a new LookAndFeel that inherits the functionality of the original one used by the LnF framework.

17.3.1. Example 1

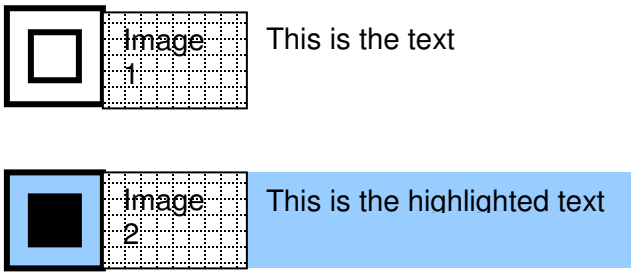
```
try
{
    // instantiate the new LookAndFeel to be used by all
    // List and ChoiceGroup options
    MyCheckboxLnF cbLnF = new MyCheckboxLnF();

    // replace the existing LnF with the new one
    LookAndFeelEngine.set(LookAndFeelEngine.LNF_ID_LCDUI_CHECKBOXITEM,
                          cbLnF);

    // since we just modified the LnF used by *ALL* Lists and
    // ChoiceGroups, we must make all the Screens of these types
    // invalid so layout calculation happens and the new LnF gets used
    LookAndFeelEngine.markAsValid(myList, false);
    LookAndFeelEngine.markAsValid(myForm, false);
}
catch (LookAndFeelException lnfe)
{
    // a problem was encountered
}
```

17.3.2. Example 2

The following example modifies the geometry of the options of a ChoiceGroup or List. The standard implementation produces the following radio-buttons:



This example will produce the following radio-buttons:



```
public class MyCheckboxLnF extends CheckboxItemLookAndFeel
{
    // the default constructor
    public MyCheckboxLnF() throws LookAndFeelException
    {
    }

    // returns the width occupied by the image and
    // the radio-button. The width of the TextView
    // will be set by the framework once this method
    // returns
    public int getPreferredWidth(Object[] params,
                                 int mode)
    {
        int w = 0;

        // retrieve the parameters
        TextView tv = (TextView)params[0];
        Image img = (Image)params[1];
        Font font = (Font)params[2];

        // let's use the font height as the dimension of the radio-button
        w += font.getHeight();

        // consider the width of the image
        w += img.getWidth();

        return w;
    }

    // returns the height
    public int getPreferredHeight(Object[] params,
                                  int mode)
    {
        int h;

        // retrieve the parameters
        TextView tv = (TextView)params[0];
        Image img = (Image)params[1];
        Font font = (Font)params[2];

        // the TextView should have been reformatted by the framework
        // already, so its height is accurate
        int tvh = tv.getHeight();

        // let's use the font height as the dimension of the radio-button
        int fh = font.getHeight();

        // consider which is higher: radio-button icon or text
        if (fh > tvh)
        {
            h = fh;
        }
```

```
        else
        {
            h = tvh;
        }

        // consider the height of the image
        int ih = img.getHeight();
        if (ih > h)
        {
            h = ih;
        }

        return h;
    }

    // paint the radio-button/checkbox, etc.
    public void paint(Graphics g, Object[] params,
                      int width, int height, int mode)
    {
        // don't need to clear the background, it was cleared by
        // the Displayable that contains this ChoiceGroup option


        // determine if the item is selected
        boolean selected = (mode & ITEM_LNF_STATE_SELECTED != 0);

        // determine if the item is highlighted (has focus)
        boolean focused = (mode & ITEM_LNF_STATE_HIGHLIGHTED != 0);

        // render the focused rectangle/highlighting
        if (focused)
        {
            // use the highlighting color
            g.setColor(getColor(ColorPalette.HIGHLIGHTED_FILL_COLOR));
            g.fillRect(0, 0, width, height);
        }

        // set the default foreground color
        g.setColor(getColor(ColorPalette.FOREGROUND_COLOR));

        // render the radio-button or checkbox (if applicable)
        int size = font.getHeight();
        switch (mode & ITEM_LNF_ASPECT_MASK)
        {
             case ITEM_LNF_ASPECT_RADIOBUTTON:
                 // outer rectangle
                 g.drawRect(0, 0, size-1, size-1);

                 // inner rectangle, if selected
                 g.drawRect(1, 1, size-3, size-3);
                 if (selected)
                 {
                     g.fillRect(1, 1, size-2, size-2);
                 }
```

```
                // move the origin of coords so the image
                // or TextView is next to this icon
                g.translate(size, 0);
                break;

            case ITEM_LNF_ASPECT_CHECKBOX:
                // outer rectangle
                g.drawRect(0, 0, size-1, size-1);

                // Inner mark, if selected
                // (the inner mark is an x)
                g.drawRect(1, 1, size-3, size-3);
                if (selected)
                {
                    g.drawLine(0, 0, size-2, size-2);
                    g.drawLine(0, size-2, size-2, 0);
                }
                // move the origin of coords so the image
                // or TextView is next to this icon
                g.translate(size, 0);
                break;
        }

        // render the image (if any)
        if (img != null)
        {
            g.drawImage(0, 0, img, 0);

            // move the origin of coords so the TextView
            // is next to this image
            g.translate(img.getWidth(), 0);
        }

         // render the TextView
         tv.paint(g);
    }
}
```

18. Multimedia

18.1. Overview

This chapter deals with the audio and video features of the i325. Due to the high level of device-dependant configurations allowed by JSR-135, detailed guidelines for the i325's MMA 1.1 implementation follow.

In each of the class or interface descriptions that follow, if a particular method is not listed it should be assumed that the i325 implements it as described in the Sun Javadocs for JSR-135 (MMA 1.0) or JSR-184 (MMA 1.1). If a class or interface listed in the Javadocs is not here, then it is either not implemented on the i325 or it contains no methods.

These tables provide a quick glance as the content types, controls, protocols, and media files supported on the i325. See "Tips" section on page 131 for encoding details on these media files.

Table 10. Content Types vs. Supported Media Files

	Media File Types					
	TONE	**MIDI`**	**WAV**	**AU**	**IDEN Voicenote**	**Digital Camera**
Content Type	audio/x-tone-seq	audio/mid audio/midi	audio/x-wav	audio/basic	audio/x-idenvselp audio/x-idenambe	Image/ jpeg

Table 11. Controls vs. Supported Media Files

		Media File Types				
		Tone Sequences	**MIDI`**	**WAV**	**AU**	**IDEN Voicenote**
Control	**Volume**	**Y**	**Y**	**Y**	**Y**	
	Tone	**Y**				
	Tempo		**Y**			
	Record					**Y**

Table 12. Locator Protocols vs. Supported Media Files

		Media File Types				
		Tone Sequences	**MIDI`**	**WAV**	**AU**	**IDEN Voicenote**
Protocol	**device://**	**Y**				
	file://		**Y**	**Y**	**Y**	**Y**
	http://		**Y**	**Y**	**Y**	**Y**

There are nine different system properties that can be queried using the method `System.getProperty(String key)`. There are three conditions that are required to say a device supports mixing. The i325 meets one of those conditions— a MIDI **or** Tone Sequence can play simultaneously with a WAV **or** AU. The default encoding for audio and snapshots are in bold.

Table 13. "System Properties

Key	System.getProperty(Key)
"microedition.media.version"	"1.1"
"supports.mixing"	"false"
"supports.audio.capture"	"true"
"supports.video.capture"	"false"
"supports.recording"	"true"
"audio.encodings"	"encoding=idenvselp" **"encoding=idenambe&rate=2200"** "encoding=idenambe&rate=4400"
"video.encodings"	null
"video.snapshot.encodings"	"encoding=jpeg&width=80&height=64" **"encoding=jpeg&width=128&height=96"** "encoding=jpeg&width=160&height=128" "encoding=jpeg&width=320&height=240" "encoding=jpeg&width=640&height=480"
"streamable.contents"	null

18.2. Class Description

The multimedia APIs are all located in package class javax.microedition.media.

The following is the Class Hierarchy for the multimedia API:

```
java.lang.Object
  |
  +-- javax.microedition.media.Manager
```

The following is the Interface Hierarchy for the multimedia API:

```
javax.microedition.media.Controllable
  |
  +-- interface javax.microedition.media.Player

javax.microedition.media.PlayerListener

javax.microedition.media.Control
  |
  +-- javax.microedition.media.control.ToneControl
  |
  +-- javax.microedition.media.control.VolumeControl
  |
  +-- javax.microedition.media.control.TempoControl
  |
  +-- javax.microedition.media.control.RecordControl
  |
  +-- javax.microedition.media.control.VideoControl
```

18.3. Method Descriptions

18.3.1. Manager Methods

18.3.1.1. createPlayer

Creates a `Player` from an input locator.

```
public static Player createPlayer (String locator)
   throws IOException, MediaException
```

For music content stored in the JAR file or in a RMS database, use the protocol `file://`. See also the "Locator Protocols vs. Supported Media Files" table on page 127.

```
public static Player createPlayer (InputStream  stream,
   String type) throws IOException, MediaException
```

The `type` will be checked against the known accepted content-types. However, during the realize process the actual content of `stream` will be analyzed for compatible data.

```
public static Player createPlayer (DataSource source)
   throws IOException, MediaException
```

This method will always throw `MediaException`. See "javax.microedition.media.protocol Tips" section on page 132.

18.3.1.2. getSystemTimeBase

This method will always return null.

18.3.1.3. playTone

Plays a tone specified by a note and duration.

```
public static void playTone (int note, int duration, int volume)
   throws MediaException
```

The i325 limits durations to a maximum of 4 seconds, but does not throw an exception if `duration` is longer. Likewise, the minimum limit for `duration` is 2 milliseconds. Similarly, the maximum and minimum limits on `note` are 103 and 63. This method throws a `MediaException` if any `Player` object is in the `STARTED` state or a previous tone is still playing.

18.3.2. Player Methods

18.3.2.1. prefetch

Acquires resources and processes as much data as necessary to reduce the start latency.

```
public void prefetch() throws MediaException
```

An exception is thrown if:

- the handset is in Vibe-All mode,
- Java does not have focus, or
- another `Player` object of this media-type is already in the `PREFETCHED` state.

Keep in mind this other prefetched `Player` may be from another MIDlet or MIDlet suite. See also 18.4.2.

18.3.2.2. start

Starts the `Player` as soon as possible

```
public void start() throws MediaException
```

If playback does not start it may be due to the device sending a `DEVICE_UNAVAILABLE` event after the `Player` was prefetched. Avoid calling `start()` in a MIDlet's `startApp()` method as this may cause the MIDlet's UI to freeze at MIDlet starting or MIDlet resuming screens.

18.3.2.3. stop

Stops the Player.

```
public void stop() throws MediaException
```

Pause and resume functionality is not supported on the i325. Stopping any media type means a subsequent call to `start()` will play from the beginning of the file.

18.4. Tips and Code Examples

See Sun's JSR-135 Javadocs for more use-case scenarios and code examples.

18.4.1. Basic Playback

On the i325 the simplest way to begin playback is the following. It assumes that the MIDI file "abc.mid" is in the root path of the JAR file.

```
{
      Player p = Manager.createPlayer("file://piano_solo.mid");
      p.start();
}
```

18.4.2. Common Mistake

Try to avoid the mistake of reusing the same Player object without first calling its `close()` method. It can cause hours of debugging frustration.

```
{
      Player p = Manager.createPlayer("file://xyz.mid");
      p.start();

      // Object "p" could still be playing. Must call p.close()
      // before reusing. Besides, there would no longer be a
      // reference. to the above midi and eventually, the maximum
      // number of Player objects in REALIZED state or beyond
      // will be reached.
      p.close();

      //Reuse "p"
      p = Manager.createPlayer("http://www.soap108.com/abc.mid");

      p.start();  //  <-- Note: On i325, an exception would have been
                  //      thrown during prefetch() if p.close()
                  //      had not been called.
                  //      See bullet #3 in section 18.3.2.1.
}
```

18.5. Compiling & Testing MMA MIDlets

Use Sun's Wireless Tool Kit 2.0 to compile code and package JAR / JAD files.

18.6. Tips

18.6.1. General Tips

- Voicenote files are played according to voice volume, not Java volume.
- The packages `com.motorola.midi` and `com.motorola.iden.voicenote` are deprecated.
- At most, 10 `Player` objects can be in the `REALIZED` state or beyond. One of the 10 would have to be closed before an eleventh could be realized.
- MIDI files may be either Type-0 or Type-1. The SP-MIDI format is also supported, but note that the i325 allows for up to 16 instruments at once.
- WAV files must be 8 kHz, 8-bit, mono PCM encoded files, formatted as a RIFF WAV file with little-endian byte order. RIFX WAV files that support the big-endian byte-ordering scheme are not recognized. In the file's 44-byte header, the "format" data must be 16 bytes in length. This follows the canonical WAV format. Note that Windows WAV files often have 18 bytes of "format" data. If the i325 cannot realize a Player with such a file, simply remove the two extra bytes. A PC tool that can automatically strip out these bytes and ensure a canonical format is at http://www.bluechillies.com/browse/W/B/H/. The application name is "StripWav 2.0.3".
- AU files must be 8 kHz, 8 bit, mono u-law encoded files. Only big-endian AU files with a magic number of ".snd" are considered. The magic field name identifies files as a Next/Sun sound file and is used for type checking and byte ordering information.
- When using the camera accessory, the baud rate must be set to Auto. On the i325, this option is found in the Main Menu under Settings > Advanced > Baud Rate.
- The first time `getSnapshot()`, `setRecordLocation()`, or `setRecordStream()` is called, a system security screen pops up and prompts the user to grant or deny. This screen suspends the MIDlet and causes all Players to go to `REALIZED` state. Once the selection is made, the MIDlet resumes but the above methods throw an exception because the `Player` is no longer in the right state. The `Player` will need to be started again and another call to this method will be needed. Thus, if permission is granted, it is highly encouraged that "one shot" not be used. Always select "blanket" or "session."

18.6.2. PlayerListener Tips

The following events are not used on the i325: `BUFFERING_STARTED`, `BUFFERING_STOPPED`, `DURATION_UPDATED`, and `STOPPED_AT_TIME`. No OEM events are implemented.

18.6.3. VolumeControl Tips

The lower the volume the more likely the audio will sound scratchy. By default, a `Player`'s volume is set to 100% of the Java Volume. See the "Acronyms and Definitions" section on page 7.

18.6.4. ToneControl Tips

The effective duration of a note should be at most 2.68 seconds. If not, the note will be clipped to this limit. `SILENT` notes are not held to this limit. On a PC, this duration limit will not necessarily exist.

18.6.5. TempoControl Tips

Only `TempoControl` is implemented on the i325 (for MIDI files only), despite inheriting methods from `RateControl`. All `RateControl` methods return –1. The range of values supported for `setTempo()` are 10 - 300 beats per minute.

18.6.6. RecordControl Tips

This control can be used to capture iDEN Voicenote Files. Recordings to a location beginning with `"file:/"` are put in a temporary folder and are removed when the application exits. To preserve such a recording it must be copied to a permanent location.

18.6.7. VideoControl Tips

The method `getSnapshot()` may be used to take a picture if the camera accessory is connected. See also "video.snapshot.encodings" in the "System Properties" table on page 128. Its `imageType` parameter is ignored as the tokens, such as width and height, are checked during `createPlayer()`.

18.6.8. javax.microedition.media.protocol Tips

The i325 does not provide any OEM data sources.

19. Lighting

19.1. Overview

The Lighting API lets a MIDlet turn on and off various lights on the phone.

19.2. Class Description

The API for the Lighting API is located in package com.mot.iden.multimedia

```
java.lang.Object
 |
 + - com.mot.iden.multimedia.Lighting
```

19.3. Method Description

19.3.1. Lighting Methods

19.3.1.1. setLight

Sets the specified light to the specified state

```
public static void setLight(int light, int state)
   throws IllegalStateException
```

The table below lists the valid values for light and state. Note that the valid values for state depend on the value of light.

Table 14. Valid Values for light and state

Light	State
LIGHT_DISPLAY	LIGHT_STATE_ON LIGHT_STATE_OFF
LIGHT_KEYPAD	LIGHT_STATE_ON LIGHT_STATE_OFF
LIGHT_STATUS	LIGHT_STATE_OFF LIGHT_STATE_RED LIGHT_STATE_GREEN LIGHT_STATE_AMBER
LIGHT_CALL_INDICATOR	LIGHT_STATE_OFF LIGHT_STATE_RED LIGHT_STATE_GREEN LIGHT_STATE_BLUE LIGHT_STATE_YELLOW LIGHT_STATE_MAGENTA LIGHT_STATE_CYAN LIGHT_STATE_WHITE

If you pass an invalid value for either light or state, this method throws an IllegalStateException

Note that the i325 does not have a status light. If you pass `LIGHT_STATUS` as the value for light, this method changes the state of the call indicator light instead. This lets you continue to run older applications that use the status light without needing to modify them.

19.3.1.2. javaOverRideTimer

Gives an application complete control of the device's lights.

```
public static void javaOverRideTimer(boolean state)
```

If state is true, this application is totally responsible for managing the device's lights while the application has focus and is in control of the display. The phone itself does not change the state of the lights.

If state is false, then the phone controls the state of the lights. The application can change the state of a light with `setLight()`, but the phone can change it at any time.

Note that this setting is reset to false when an application loses focus on a display. If the application wishes to override this setting, it has the opportunity to do so when it regains control of the display again.

19.3.1.3. getPhotoSensorLevel

Returns the current amount of ambient light according to the photo sensor.

```
public static int getPhotoSensorLevel()
```

The returned value ranges from 0 to 255, with 0 being no ambient light and 255 being a great deal of ambient light.

19.3.2. Deprecated APIs

The following APIs have been deprecated since the release of the i95cl.

```
public static void backlightOn()
public static void backlightOff()
public static void keypadLightOn()
public static void keypadLightOff()
public static void setStatusLight(int color)
```

19.4. Tips

- When an application uses the Lighting API and does not request that it override the native ergonomic settings, then the state of the light may appear to do strange things. An example of this is the photo sensor. Say the photo sensor is turned on and a user hits a key, which triggers the photo sensor light to turn off. Then say the light in the room is low and the photo sensor light turns back on when the user didn't press any key. It is recommended if you're going to be doing more then just flashing the lights, override the light settings to prevent the lights from changing states unexpectedly.
- When overriding the lights, none of the native ergonomic battery savings for powering off the lights is in effect and great care should be taken. Leaving all of the lights on for extended periods of time can drain the battery of the phone quickly.

20. Vibrator API

20.1. Overview

The Vibrator class lets a MIDlet turn the phone's vibrator on and off. It also provides the user with re-occurring effects that can be used with the vibrator. These effects allow the vibrator to turned on and off in reoccurring patterns. This feature is useful for games or alarms.

Note that the vibrator can be turned on for a maximum of 500 ms at any given time, and it must remain off for at least 50 ms before being turned it back on. This duty cycle is enforced in the API. These are important for periodic vibration since these constraints can affect how a MIDlet can use this class.

20.2. Class Description

The API for Vibrator is located in package com.mot.iden.multimedia. This class contains various multimedia classes like the Vibrator API.

```
java.lang.Object
 |
 + -- com.mot.iden.multimedia.Vibrator
```

20.3. Method Descriptions

20.3.1. Vibrator Methods

20.3.1.1. vibrateFor

Turns on the vibrator for the specified amount of time.

```
public static void vibrateFor(int timeOnInMs)
```

`timeInMs` is amount of time in milliseconds to vibrate the phone.

20.3.1.2. vibratePeriodicaly

Turns the vibrator on and off repeatedly.

```
public static void vibratePeriodicaly(int timOnInMS)
```

This method continuously turns the vibrator on and off for equal amounts of time. `timeOnInMS` is both the amount of time the vibrator is turned on and the amount of time it's turned off.

To stop the vibrator from turning on and off, call `vibratorOff()`.

```
public static void vibratePeriodicaly(int timeOnInMS,
   int timeOffInMS)
```

This method continuously turns the vibrator on for one amount of time and turns it off for another amount of time. `timeOnInMs` is the amount of time to turn the vibrator on in milliseconds. `timeOffInMs` is the amount of time to turn the vibrator off in milliseconds.

To stop the vibrator from turning on and off, call `vibratorOff()`.

20.3.1.3. vibratorOff / vibratorOn

The following methods allow a MIDlet to turn the vibrator on and off:

```
public static void vibratorOff()
public static void vibratorOn()
```

`vibratorOff()` stops the vibrator. `vibratorOn()` turns the vibrator on for `MAX_VIBRATE_TIME`, 500ms.

20.4. Code Examples

20.4.1. Example 1

The example below will vibrate the phone for 300 milliseconds

```
public void vibratePhone()
{
      /* this will have the phone vibrate for 300ms */
   Vibrator.vibrateFor(300);
}
```

20.4.2. Example 2

The following example allows the vibrator to vibrate periodically for 300ms using the `vibrateFor()` and `Thread.sleep()` methods:

```
public void vibratePhone()
{
      while(true){
            /* this will have the phone vibrate for 300ms */
            Vibrator.vibrateFor(300);

            /* have the phone rest for 300 ms */
            Thread.sleep(300);
      }
}
```

While this works great, the above example would tie up the execution thread. Thus, using the `vibratePeriodicaly()` method is ideal for this example.

```
public void vibratePhone()
{
   /* this will have the phone vibrate for periodically 300ms */
      Vibrator.vibratePeriodicaly(300)
}
```

20.5. Tips

Vibrating the phone drains the battery. To extend the battery life, limit the use of the vibrator.

20.6. Emulator Stub Classes

When a MIDlet uses the Vibrator class, it prints the action being performed on the transcript window. For example if a MIDlet turns on the vibrator for 10 milliseconds, "`Vibrator Turned On`" is displayed on the window.

21. Customer Care API

21.1. Overview

The Customer Care API lets J2ME™ applications access unit and user specific data. This data may be used to track and troubleshoot issues out on the field. Specifically it will provide access to such as the unit info, system status, reset/error log, client info, my info, and Java System metrics. This feature is protected with the "System Information Access" or "Read User Data Access" functional groups of the Permission/Security Domain feature, which is described in “MIDP 2.0 Security API on page 50.

21.2. Class Description

The API for the Customer Care is located in package com.mot.iden.customercare

```
java.lang.Object
 |
 + -- com.mot.iden.customercare.CustomerCare
```

21.3. Method Descriptions

21.3.1. CustomerCare Methods

21.3.1.1. getUnitInfo

Returns information about this phone such as the phone model, the version of the codeplug, the version of the CSD, the version of the software, the GPS version, and the version of the user file.

```
public static final String getUnitInfo(int fieldID)
   throws IllegalArgumentException
```

`fieldID` must be one of the values in this table:

fieldID	Example of Data
DEVICE_MODEL	"i90cA"
CP_VERSION	"19.00/19.00"
CSD_VERSION	"C97.05.05"
SW_VERSION	"D76.01.09"
GPS_VERSION	"SiRF Cust SW Version Info"
USR_VERSION	"U00c.00.00"

If `fieldID` is not one of those values, this method throws an IllegalArgumentException.

21.3.1.2. getSystemStatus

Returns system status information such as signal quality, carrier channel, carrier color code (including extended color code for supporting devices), power cutback level and serving cell quality.

```
public static final String getSystemStatus(int fieldID)
   throws IllegalArgumentException
```

`fieldID` must be one of the values in this table:

fieldID	Example of Data
SQE	"34.73"
CARRIER_CHNL	"2EB"
COLOR_CODE	"1"
PWR_CUTBACK	"00db"
SVG_CELL_QUALITY	"-66".

If `fieldID` is not one of those values, this method throws an IllegalArgumentException.

21.3.1.3. getSystemInfo

Returns system information such as the total Program and Data Space and the available Program and Data Space

```
public static final int getSystemInfo(int fieldID)
   throws IllegalArgumentException
```

`fieldID` must be one of the values in this table:

fieldID	Example of Data
DATA_SPACE_FREE	3143
DATA_SPACE_TOTAL	3124
PROG_SPACE_FREE	3964
PROG_SPACE_TOTAL	3968

If `fieldID` is not one of those values, this method throws an IllegalArgumentException.

21.3.1.4. getMyInfo

Returns information on the various service ids for the device, such as private dispatch id (as a '*' delimited UFMI), the main phone number, the alternate line phone number for supporting devices, and the carrier IP address

```
public static final String getMyInfo(int fieldID)
   throws IllegalArgumentException
```

`fieldID` must be one of the values in this table:

fieldID	Example of Data
PRVT_ID	"902*43*12345"
LINE_1	"5551234567"
LINE_2	"5558901234"
CARRIER_IP	"123.45.67.89".

If `fieldID` is not one of those values, this method throws an IllegalArgumentException.

For all `fieldIDs`, a null will be returned for applications in unauthorized domains.

21.3.1.5. getClientInfo

Returns client information such as a unique identifier (or IMEI), the device's serial number, and the SIM identifier.

```
public static final String getClientInfo(int fieldID)
   throws IllegalArgumentException
```

`fieldID` must be one of the values in this table:

fieldID	Example of Data
IMEI	"01010101010101"
SERIAL_NUMBER	"1234567890"
SIM_ID	"12345678901234

For `IMEI`, `SERIAL_NUMBER`, and `SIMID`, the device provides hashed unique values so that applications can identify the phone with a unique identifier

21.3.1.6. getErrors

Returns information on the errors that have occurred on the device.

```
public static final String getErrors(int fieldID)
   throws IllegalArgumentException
```

The errors are kept in the reset log. This log stores only the last 24 errors.

`fieldID` must be one of the values in this table:

fieldID	Example of Data
RESET_NUMBER	"1", "2"
RESET	"R[0]:R0400Date:N/AX-&1019594CT100CFC", "R[0]:RXXXXDate:XX/XX&XXXXXXXXXXXXXXX R[1]:RXXXXDate:XX/XX&XXXXXXXXXXXXXXX"

If `fieldID` is not one of those values, this method throws an IllegalArgumentException.

21.4. Code Examples

```
public void test(){

        try {
         //  Get unit info methods
         //  1) device model
         //  2) codeplug version
         //  3) csd version
         //  4) software version
         //  5) gps version
         //  6) usr version
            for(int x = 1; x < 7; x++){
                System.out.println("Unit Info Methods(" + x + ") ->" +
                    CustomerCare.getUnitInfo(x));
                screen.append("Unit Info Methods(" + x + ") ->" +
                    CustomerCare.getUnitInfo(x));
            }
```

```
//  Get system status methods
//  7) sqe signal quality
//  8) carrier channel
//  9) carrier color code
// 10) power cutback level
// 11) serving cell quality
   for(int x = 7; x <12; x++){
       System.out.println("System Status Methods(" + x + ") ->" +
           CustomerCare.getSystemStatus(x));
       screen.append("System Status Methods(" + x + ") ->" +
           CustomerCare.getSystemStatus(x));
   }

// Get my info methods
// 12) private ID
// 13) line 1 phone number
// 14) line 2 phone number
// 15) carrier IP address
   for(int x = 12; x <16; x++){
       System.out.println("My Info Methods(" + x + ") ->" +
           CustomerCare.getMyInfo(x));
       screen.append("My Info Methods(" + x + ") ->" +
           CustomerCare.getMyInfo(x));
   }

// Get client info methods
// 16) imei number
// 17) serial number
// 18) sim ID
   for(int x = 16; x <19; x++){
       System.out.println("Client Info Methods(" + x + ") ->" +
           CustomerCare.getClientInfo(x));
       screen.append("Client Info Methods(" + x + ") ->" +
           CustomerCare.getClientInfo(x));
   }

// Get error/reset methods
// 19) reset number
// 20) reset log
   for(int x = 19; x <21; x++){
       System.out.println("Reset & Error Methods(" + x + ") ->" +
           CustomerCare.getErrors(x));
       screen.append("Reset & Error Methods(" + x + ") ->" +
           CustomerCare.getErrors(x));
   }

// Get java system methods
// 21) free data space
// 22) total data space
// 23) free program space
// 24) total data space
   for(int x = 21; x <25; x++){
       System.out.println("Java System Methods(" + x + ") ->" +
           CustomerCare.getSystemInfo(x));
```

```
                screen.append("Java System Methods(" + x + ") ->" +
                    CustomerCare.getSystemInfo(x));
            }

        }
        catch(Exception e){
            System.out.println("Something went wrong! "+ e.toString());
        }
    }
```

21.5. Compiling & Testing Customer Care MIDlets

In the stub classes, the methods return either 0 or null since there is no native support for them.

22. Java ZIP

22.1. Overview

The Java Zip API has been included as an enhancement especially well suited for a limited bandwidth data device such as an iDEN phone. It allows for file deflation before sending data via the network and inflation after receiving data from the network to best use the bandwidth available. Downloading a zipped file and then decompressing on device is usually much faster than downloading uncompressed content.

The Java ZIP API consists of ZipEntry, ZipInputStream, ZipOutputStream, and ZipException. This package provides classes for reading and writing data in the standard ZIP (WinZip archive) format.

The Java ZIP API is simply compatible with the Sun's J2SE™ v1.4 ZIP API (java.util.zip). Please refer to the following web page for details: http://java.sun.com/j2se/1.4.2/docs/api/index.html.

22.2. Class Description

The API for the ZIP is located in the com.mot.iden.zip package.

```
java.lang.Object
 |
 + - com.mot.iden.zip.ZipEntry
```

22.3. Method Descriptions

Please refer to the relevant Javadocs (java.util.zip).

22.4. Code Example

22.4.1. Get ZipEntry information

```
public static void print(ZipEntry e)
{
      PrintStream err = System.err;
      err.print("added " + e.getName());
      if  ( e.getMethod() == ZipEntry.DEFLATED ) {
            long size = e.getSize();
            if (size > 0) {
                  long csize = e.getCompressedSize();
                  long ratio = ((size-csize)*100) / size;
                  err.println(" (deflated " + ratio + "%)");
            } else {
                  err.println(" (deflated 0%)");
            }
      } else {
            err.println(" (stored 0%)");
      }
}
```

22.4.2. ZipOutputStream/ZipInputStream

```
try {
      ByteArrayOutputStream gis = new ByteArrayOutputStream(1024);

      // (1) Compression: Define ZIPOutputStream with
      //     ByteArrayOutputStream
      ZipOutputStream os = new ZipOutputStream(gis);

      ZipEntry zipentry = new ZipEntry("TEST1");
            /* set the 1st entry name */
      os.putNextEntry(zipentry);

      // (2) Writes the string to the ZIPOutputStream
      os.write("This chapter covers how to configure a system "+
            "without a name service. Administration is ...".getBytes());

      zipentry = new ZipEntry("TEST2");
            /*  set the 2nd entry name  */
      os.putNextEntry(zipentry);

      // (3) Writes the string to the ZIPOutputStream
      os.write("The document you requested is not found. " +
            "It may have expired or moved.".getBytes());

      os.close();

      // (4) Decompression: Get input compressed data
      //     from the gis stream
      ByteArrayInputStream gis1 =
            new ByteArrayInputStream(gis.toByteArray());

      // (5) Define ZIPInputStream with ByteArrayInputStream
      ZipInputStream os1 = new ZipInputStream(gis1);

      byte[] buf1 = new byte[2048]; /* Decompressed buffer */
      int ch;
      ZipEntry entry;

      // (6) Reads the compressed stream to the decompressed buffer
      while ((entry = os1.getNextEntry()) != null) {
            System.out.println("Extracting: " + entry);

            while ((ch = os1.read(buf1, 0, buf1.length - 1)) >= 0) {
                  System.out.println(new String(buf1, 0, ch));
            }
      }

       os1.close();

} catch (Exception e) {
      e.printStackTrace();
}
```

23. Smart Text Entry

23.1. Overview

Text components on the i325 phone are enabled with T9 smart text entry capability. Smart text entry allows users to enter text faster by removing the need for multiple key presses in order to input certain letters or symbols. The T9 engine recognizes given key sequences and matches them with the words contained in its database. Because multiple words may be entered with a single key sequence, users can access other matches in the word database, or explicitly enter any word character by character.

LCDUI based MIDlets that conform to the MIDP specification and use TextField or TextBox objects and LWT based MIDlets that use TextField or TextArea objects automatically benefit without any additional coding effort.

The rest of this section explains various T9 features and how users interact with T9 in a MIDlet.

23.2. T9 Features

The T9 engine allows users to enter text in one of four different ways: Word entry mode, Alpha entry mode, Numeric entry mode, and Symbol entry mode. Additionally, the T9 engine supports multiple languages, which can be combined with Word entry mode to enter text in English, Spanish, French, Portuguese, Hebrew, or Korean.

iDEN phones prior to the i325 phone relied only on multi-tap text entry in text components. With multi-tap entry, a user is required to press a key multiple times to access one of the multiple alphabetic characters mapped to that key. For example, to enter the character 'c', the '2' key must be pressed 3 times.

Text components can still use this method of entry, which is also called Alpha mode. While Alpha mode is not the most efficient way to enter characters on the phone, it is sometimes necessary in order to enter words that are not in the T9 word database.

For words that are in the database, T9's Word mode is much more efficient. With Word mode, users are not required to explicitly enter each character for a word. For example, without Word mode the word "back" requires 8 key presses to enter- double the amount of letters. With Word mode, a user needs to press only 4 keys. The T9 engine recognizes the key sequence as the word "back" since that is the only possible match for the sequence. In cases where there is more than one match, the user can press the "0" key, also labeled "next", to match to the next word. If a desired word is not in T9's word database, users can switch to Alpha mode and enter the word explicitly.

In cases where only numeric characters need to be entered, T9's "Numeric" mode is used. This mode simply maps the keys pressed to their labeled numeric value.

While users can access symbols like punctuation through Alpha and Word modes, T9 also has a "Symbol" mode. The Symbol mode operates slightly differently than the other modes because symbol entry is done on a separate screen. When the user is done choosing all the symbols needed, those symbols are inserted into the text component at the current cursor position.

23.3. The T9 UI

A MIDlet's UI is automatically integrated with the T9 UI when a text component is present and has focus. While entering text into a text component, the T9 UI consists only of the icon representing the current entry mode and any capitalization taking place and word highlighting.

In order to maximize the amount of screen space on the device, the space for the entry mode icon is now shared with the menu icon. If the MIDlet has more than two Screen Commands or if the TextField has an Item Command, the menu icon is displayed in place of the entry mode icon.

A T9-Enhanced MIDlet

The screenshot on the left displays a typical email MIDlet. The focused text component is the Message box.

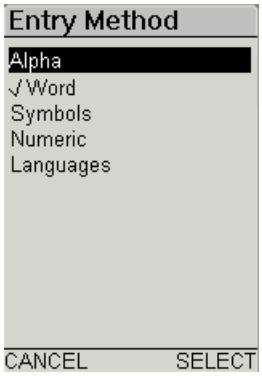
The text within the Message box is being edited in Word mode, indicated by the icon at the bottom center of the screen and the word "application" is the returned word based on the key sequence entered. The user has pressed the '#' key enabling character shift, indicated by the up arrow in the icon. The next character entered by the user will be returned as a capital letter.

23.4. Changing T9 Entry Mode

Smart text entry in a MIDlet is nearly seamless, giving the MIDlet more functionality without complicating it. However, users can interact in a more direct manner with the T9 engine to switch entry modes and change language. Although a user's interaction with the T9 engine is completely transparent to the MIDlet, users are required to go through a MIDlet's UI in order to access the T9 UI.

In both LWT and LCDUI based MIDlets the T9 entry mode screen is accessed by pressing the Menu key. If a MIDlet has more than two Screen Commands or the TextField has an Item Command, the first menu key press accesses the Command menu screen per the MIDP specification. If the MIDlet was previously focused on a TextField, the second menu key press accesses the T9 "Entry Method" menu. For MIDlets with two or fewer Screen Commands and no Item Command to access, the menu key automatically accesses the Entry Method menu when focused on a TextField.

The image to the right shows how the Entry Method menu appears in an LCDUI based MIDlet. This menu allows the user to select the desired mode of text entry. Selecting the Alpha, Word, or Numeric modes returns the user to the last MIDlet screen and changes the entry mode of the focused text component. Selecting Language or Symbol brings the user to a different screen where they can select the desired language or symbols.



The Entry Methods

23.5. Influencing T9

Although there is no code change required to take advantage of T9 in an LCDUI or LWT based MIDlet, there are ways to influence T9 behavior.

The MIDP 2.0 specification has been updated to include APIs that allow developers to directly control the smart text engine for a TextField. These APIs are capable of controlling entry mode, capitalization, and language among other features. Refer to the MIDP 2.0 specification for more information on those APIs.

In addition to receiving direction with the MIDP 2.0 APIs, the smart text engine is affected by implicit settings on the phone. Just as a MIDlet can take advantage of the internationalization features of the i325 phone by displaying the MIDlet name in different languages, the T9 engine detects the default language of the user and starts itself in the appropriate language setting.

The initial entry mode of the T9 engine can also be affected by the constraints of the text component that it services. For example, a TextField created with the `TextField.PHONENUMBER` constraint will create a TextField with a T9 engine set initially to Numeric entry mode. In addition, the user will not be able to access the Entry Method menu to change the input mode. Similarly, a TextField created with the `TextField.NUMERIC` constraint will initially be set to Numeric entry mode but will also allow for negative numbers. Text components whose constraints are set to `NUMERIC` or `PHONENUMBER` after instantiation will change their entry mode accordingly, but relaxing constraints will not change the entry mode of a text component.

23.6. T9 Engine Lifecycle

Text components each have their own instance of the T9 engine, allowing MIDlets to contain multiple text components at once, each performing different smart text functions and using different entry modes.

MIDlets that reuse text components on different screens typically clear out any old text before redisplaying that component to a user. While the MIDP API provides for such functionality, it does not provide for any functionality to reset the T9 engine of a text component. MIDlets that reuse text components will also carry the old T9 state of that text component when it is redisplayed. For example, if a user last left a TextField in Alpha entry mode and that TextField was removed from one Screen object and added to another, it will look as if it was initialized in Alpha mode in the new Screen. To avoid this behavior, text components should not be recycled.

Appendix A. Fonts on the i325 Phone

A.1. Overview

As MIDP states, the Font class represents fonts and font metrics. Fonts cannot be created by applications. Instead, applications query for fonts based on font attributes and the system attempts to provide a font that matches the requested attributes as closely as possible. A Font's attributes are style, size, and face. The style value may be combined using the OR operator whereas the size and face attributes cannot be combined.

A.2. Available Fonts

The i325 phone offers 3 different sizes, styles, and faces. The table describes the font faces:

Table 15. Font Faces

Name	Description
FACE_PROPORTIONAL	Each font has a variable width and a fixed height.
FACE_MONOSPACE	Each font has a fixed width and height.
FACE_SYSTEM	Each font has a variable width and a fixed height. These fonts are used in the ergonomics of the i325 phone.

This table describes the font sizes:

Table 16. Font Sizes

Name	Description
SIZE_SMALL	The i325 phone only offers small size fonts for FACE_MONOSPACE faces.
SIZE_MEDIUM	The i325 phone offers medium size fonts for all 3 font faces.
SIZE_LARGE	The i325 phone only offers large size fonts for FACE_MONOSPACE font faces.

This table describes the font styles:

Table 17. Font Styles

Name	Description
STYLE_PLAIN	The i325 phone offers plain style fonts for all 3 font faces and sizes.
STYLE_UNDERLINED	The i325 phone offers underlined style fonts for all 3 font faces and sizes.
STYLE_BOLD	The i325 phone offers bold style fonts for all 3 font faces and sizes.
STYLE_ITALIC	The i325 phone does not offer any italic font styles.

A.3. Default Fonts

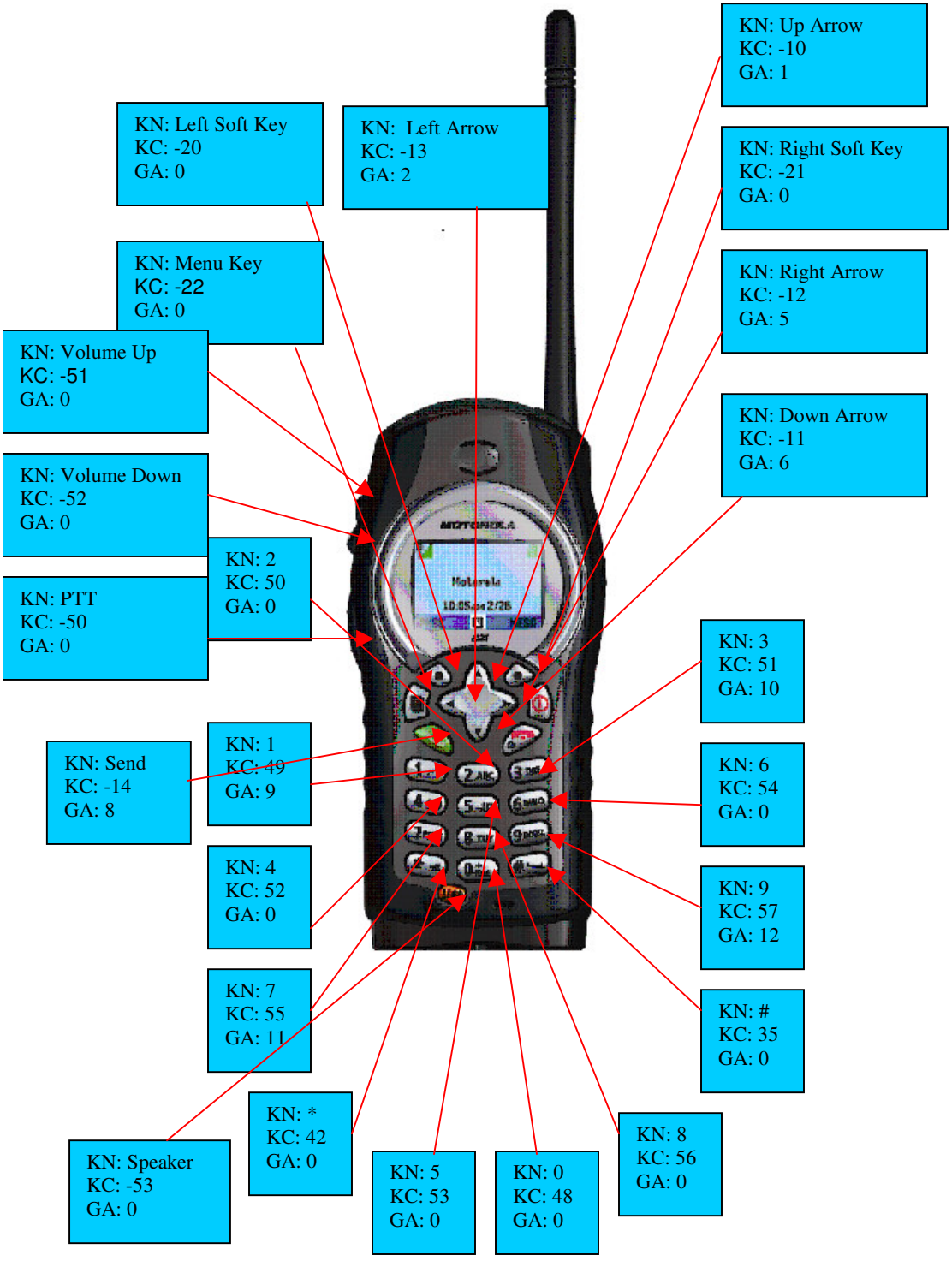
The default font is set to `FACE_SYSTEM`, `SIZE_MEDIUM`, and `STYLE_PLAIN`. LCDUI components use the default font class from the com.motorola.iden.lnf package. (See "

MIDP 2.0 LCDUI page 28.) The font face and size for LCDUI components defaults to `FACE_SYSTEM`, `STYLE_PLAIN`, while the size depends on the user's selected font size in the phone's settings.

A.4. Legacy Fonts

The monospace font offered in the i325 is the same font from previous Java-enabled iDEN phones. The proportional font is the same font from the i85s and i88s phones. The system font is a completely new font for the i325 sharing only the same font height as previous Java-enabled iDEN phones' system fonts.

Appendix B. Key Mapping Of The i325 Phone



Appendix C. How To

C.1. Downloading to the Device

The JAL utility provides the application developer a means to load an application to the i325. Additionally, a data cable is used to connect to the bottom connector on the i325.

For instructions on installing the JAL utility, please consult the user guide. In order for JAL to communicate with the i325, the data-communication rates need to be in sync. The following instructions describe the steps required to change the data-communication rate of the i325 phone:

1. From the main menu, select "Settings"
2. From the "Settings" menu, select "Advanced"
3. From the "Advanced" menu, scroll down to the "Baud Rate" menu item
4. Select "Change" from the command choices
5. Scroll down to the desired baud rate, and press the "Select" soft key

NOTE:	Begin by setting the Baud Rate to 19200 to ensure the JAL utility can communicate with the device. If a Baud Rate of 19200 works, try increasing the speed incrementally.

For information on configuring and using JAL, refer to the users guide.

After loading the JAR and JAD file on the i325, the friendly names specified in the MANIFEST.MF file for the MIDlets should appear on the Java menu. For the HelloWorld example, the Java menu will contain an item "HelloWorld" representing the application. At this point, the application is only "Loaded" on the i325 and not yet installed. From this point, the application may either be installed or removed from the device.

NOTES:	The number of MIDlet suites that can be installed on the i325 is limited to 32. If the number of MIDlets suites installed is not less than 32, de-install an application before proceeding. The application must be installed before it can be executed. The following steps describe the installation procedure

C.2. Installation

The following checklist should be covered before attempting to install a MIDlet suite. Failure to verify this checklist could lead to an installation failure.

- Application supports CLDC 1.0 and MIDP 2.0 (the configuration and profile supported by the i325)
- A JAD file has been created
- A JAR file containing META-INF/MANIFEST.MF has been created
- The MIDlet-Name, MIDlet-Version, and MIDlet-Vendor attributes are duplicated in both the MANIFEST.MF and the JAD file
- The total size of the JAR file is < 300KB
- Both the JAD and JAR file have the same name (except for the .jad and .jar extensions)
- File names are less than 16 characters (including extension)
- Less than 32 MIDlet suites are currently installed
- There's enough Data and Program Space
- The JAR size listed in JAD matches actual JAR size

NOTES: To check Program and Data space from the Java menu, select "Java System" and press the "Select" Soft key.

Class files are stored in Program Space.

JAR files are stored in Data Space before installation. After installation, the JAR file is destroyed.

Resource files are stored in Data Space after installation.

To install the MIDlet suite, highlight the suite in the Java menu, and press the "Install" soft key. A dialog is displayed indicating the device is currently installing the application. The dialog indicates exactly the steps being executed.

NOTES: **Java Application Installer/De-Installer (JAID)**
JAID is a component built into the Motorola KVM to handle installing and uninstalling Java applications on a device. The process of installing an application is time intensive. It involves loading the class files from the JAR file and writing the image, in a platform-specific manner, to memory. By installing Java applications, class files do not have to be stored in RAM, allowing more runtime memory for the application at hand. Additionally, the time required to launch Java applications is decreased dramatically.

After successful installation, the class files are placed in the Program Space and the resource files are placed in the Data Space. The original JAR file is then destroyed.

Applications need to be JAID installed only once. If the i325's software is upgraded, Java applications must be re-installed.

Once the application is done installing on the i325, you need to press the "Done" soft key to return to the Java Menu.

If you leave the installation progress screen while the MIDlet suite is still being installed, then you return to the installation screen the next time you select Java Apps from the Main Menu.

C.3. To Remove the Application *Before* Installing or After an Installation Failure

1. Highlight the application on the Java menu.
2. Press the "Menu" key. A new menu is then displayed.
3. Highlight the "Deinstall" menu choice and press the "Yes" soft key.
4. The Java menu re-appears when competed. The removed application is no longer present in the list.

C.4. Starting Applications

Often times a MIDlet suite contains only one MIDlet. If so, then a user can launch that MIDlet from the Java Menu simply by highlighting that MIDlet suite and pressing the "Run" soft key.

If there are multiple MIDlets in the suite, then the "Open" soft key is displayed. Once pressed, the MIDlet suite is opened and the user can highlight one of the individual MIDlets. From there, pressing the "Run" soft key launches the MIDlet.

C.5. Exiting Applications

During the development process, a MIDlet may not exit properly via the correct and elegant method. The i325's policy on Java applications is to allow the user to exit an application at anytime, either forcefully or with a menu option. If an application, during the development process, becomes unstable or fails to respond, the user or developer may end the application by pressing the home key.

The following steps outline the method to exit applications forcefully:

1. Press the Home/End Key.

   A "Suspended Apps" screen with "End" and "Back" soft keys appears.
2. To resume the application, press the "Open" soft key.
3. To end the application, press the "Menu" hard key, then Highlight the "End" menu choice, press the "Select" soft key.

C.6. Java System Formatting and Diagnosis

From the Java Apps menu, a Java System option is available to format and diagnose the Java platform on the i325. You can use these tools to diagnose the system's integrity as well as to re-format the Program and Data Space that the Java apps use. In addition to the diagnosis and formatting tools, the Java System option provides information on platform specifications and available storage space. The following table summarizes the tools available through the Java System option.

Tool	Description	Location of tool.
Java System	Provides information on platform specifications and storage space.	Highlight the "Java System" menu choice from the Java Apps menu, and press the "Select" soft key.
Reset System	Checks integrity of Program and Data Space. Also re-associates JAD and JAR files.	Highlight the "Java System" menu choice from the Java Apps menu, and press the "Menu" soft key. Highlight the "Reset System" menu choice and press the "Select" soft key.
Delete All	Removes all applications and resources used by Java in both the Program and Data space.	Highlight the "Java System" menu choice from the Java Apps menu, and press the "Menu" soft key. Highlight the "Delet All" menu choice, and press the "Select" Soft key.

Appendix D. Internationalization Support

D.1. Country Codes

The following table displays the ISO-3166 Country Code selected for the user specified Country Alias:

Country Alias	Converted ISO-3166 Country Code	Converted ISO-3166 Country Name
"AL",	"AL"	Albania
"AND",	"AD"	Andorra
"AUS",	"AU"	Australia
"AZE",	"AZ"	Azerbaijan
"BG",	"BG"	Bulgaria
"BGD",	"BD"	Bangladesh
"BHR",	"BH"	Bahrain
"BIH",	"BA"	Bosnia & Herzegovina
"BRU",	"BN"	Brunei
"BW",	"BW"	Botswana
"CAN",	"CA"	Canada
"CH",	"CH"	Switzerland
"CHN",	"CN"	China
"CI",	"CI"	Cote D'Ivoire
"CPV",	"CV"	Cape Verde
"CY",	"CY"	Cyprus
"CZ",	"CZ"	Czech Republic
"DK",	"DK"	Denmark
"EE",	"EE"	Estonia
"EGY",	"EG"	Egypt
"ETH",	"ET"	Ethiopia
"F",	"FR"	France
"FI",	"FI"	Finland
"GEO",	"GE"	Georgia
"GH",	"GH"	Ghana
"GIB",	"GI"	Gibraltar
"GN",	"GN"	Guinea
"GR",	"GR"	Greece
"HK",	"HK"	Hong Kong
"HR",	"HR"	Croatia (Hrvatska)
"IL",	"IL"	Israel
"INA",	"IN"	India
"IND",	"IN"	India
"IRL",	"IE"	Ireland
"IS",	"IS"	Iceland
"JOR",	"JO"	Jordan
"KGZ",	"KG"	Kyrgyzstan
"LSO",	"LS"	Lesotho
"LV",	"LV"	Latvia
"MAC",	"MO"	Macau
"MDG",	"MG"	Madagascar
"MKD",	"MK"	Macedonia

"MOZ",	"MZ"	Mozambique
"MW",	"MW"	Malawi
"MY",	"MY"	Malaysia
"NAM",	"NA"	Namibia
"NCL",	"NC"	New Caledonia
"NL",	"NL"	Netherlands
"NZ",	"NZ"	New Zealand
"OMN",	"OM"	Oman
"PAK",	"PK"	Pakistan
"PH",	"PH"	Philippines
"POL",	"PL"	Poland
"QAT",	"QA"	Qatar
"RO",	"RO"	Romania
"RUS",	"RU"	Russia
"SA",	"SA"	Saudi Arabia
"SDN",	"SD"	Sudan
"SGP",	"SG"	Singapore
"SI",	"SI"	Slovenia
"SK",	"SK"	Slovakia
"SYR",	"SY"	Syrian Arab Republic
"TH",	"TH"	Thailand
"TR",	"TR"	Turkey
"TZ",	"TZ"	Tanzania
"UKR",	"UA"	Ukraine
"UZB",	"UZ"	Uzbekistan
"VN",	"VN"	Vietnam
"YU",	"YU"	Yugoslavia
"ZW",	"ZW"	Zimbabwe
"",	"US"	United States

D.2. Language Codes

The following table specifies the ISO-639 Language Codes supported by the i325 phone.

ISO-639 Code*	Language
"en"	English
"es"	Spanish
"fr"	French
"pt"	Portuguese
"he"	Hebrew
"kr"	Korean

* All languages may not be available on the handset.

Appendix E. Playing MIDI Files

The MidiPlayer API provides some basic controls for playing music. The API allows the application to start and stop playing a MIDI file and to play a single MIDI note on top of the music.

E.1. Class Description

The API for the MidiPlayer is located in package com.motorola.midi. The class MidiPlayer contains all the methods needed to work with MIDI.

```
java.lang.Object
 |
 + -- com.motorola.midi.MidiPlayer
```

E.2. Playing MIDI Files

These methods start and stop playing the specified MIDI file.

```
public static void play(String filename, int mode)
   throws IllegalArgumentException, IOException
```

`filename` is a MIDI file stored locally on the device. `mode` instructs the native MIDI engine to either play the file one time or continuously. The two possible values for `mode` are `MidiPlayer.PLAY_ONCE` and `MidiPlayer.PLAY_CONTINUOUS`. You can play only one MIDI file at a time. Calling `play()` while another MIDI file is already playing stops the old file and starts the new file.

```
public static void stop()
```

This method discontinues the playing of any MIDI files. Calling the method while a MIDI is not playing has no effect.

E.3. Playing a Note

Plays a single MIDI note on top of a playing MIDI file.

```
public static void playEffect(int note, int key_pressure,
      int channel_vol, int instrument, int channel)
      throws IllegalStateException, llegalArgumentException
```

You cannot use this method unless a MIDI file is playing.

The note has four characteristics—pitch, key pressure, volume, and instrument type—and one option—the channel to play on.

E.4. MIDI Instruments

You can find a list of instrument values that can be passed to `playEffect()` at http://midistudio.com/Help/GMSpecs_Patches.htm. Acceptable ranges for each parameter are as follows:

Parameter	Min	Max
note	0	127
key_pressure	0	127
channel_vol	0	127
instrument	0	176
channel	0	15

E.5.Code Examples

The following code demonstrates each of the three method calls.

```
try {
         MidiPlayer.play("soap108dotcom.mid", MidiPlayer.PLAY_ONCE);

         int NOTE = 60;  /* Middle C (C3) */
         int VOL = 127;  /* Maximum volume */
         int INST = 74;  /* Flute */

         MidiPlayer.playEffect(NOTE,
                               int key_pressure = 120,
                               VOL,
                               INST,
                               int channel = 10);
} catch (Exception e) {
}

MidiPlayer.stop();
```

E.6.Tips

Often it is useful to tell if a MIDI is still playing (e.g. you may want to determine length of the playing time). To do this, pass a known illegal value to `playEffect()`. The native implementation's order of checking for errors in parameters allows this code to be useful in this regard:

for errors in parameters allows this code to be useful in this regard:

```
try {
         /* Intentionally pass a parameter that is out of range.
          * There are only 16 channels available.
          */
         int INVALID_CHANNEL = 5000;
         MidiPlayer.playEffect(65, 120, 120, 0x56, INVALID_CHANNEL);
} catch (IllegalArgumentException e1) {
        /* A MIDI IS playing */
} catch (IllegalStateException e2) {
        /* A MIDI is NOT playing */
}
```

It is recommended that you don't call `play()` in `startApp()`. Applications are not allowed to play MIDI files before having the display, and an application can only get the display after `startApp()`.

MIDI files can be packaged as resources inside the JAR file. Additionally, with the File API you can write out a custom MIDI file and store it in the phone, or download one from the Internet using the networking classes.

Refer to "MIDI Instruments" on page 157 for additional information on MIDI instruments.

E.7.Compiling & Testing MIDI Capable MIDlets

The stubbed MidiPlayer class is a non-functional class. The class is provided to build and run within any emulator. The class will make an attempt to display its methods' behavior through `System.out.println()` statements when a method is called.

`play()` displays "Playing a MIDI.".

`stop()` displays "MIDI Stopped.".

`playEffect()` displays "Playing a MIDI Effect….Done".

Appendix F. Optional Attributes for JAD

Attribute Name	Attribute Description
iDEN-MIDlet-Name-<LANG>	The name of the MIDlet suite that identifies the MIDlets to the user in <LANG> language. The language codes, <LANG>, can be obtained from "Internationalization Support" on page 155.
iDEN-Vendor-<LANG>	The organization that provides the MIDlet suite in <LANG> language. The language codes, <LANG>, can be obtained from "Internationalization Support" on page 155.
iDEN-MIDlet-<LANG>-<n>	The name, icon and class of the n^{th} MIDlet in the JAR file separated by a comma in <LANG> language. The language codes, <LANG>, can be obtained from "Internationalization Support" on page 155
IDEN-MIDlet-Phone: <CLASSPATH>	The MIDlet that is capable of receiving a phone call. <CLASSPATH> must be the class path of a MIDlet that will handle the incoming call.
iDEN-MIDlet-miniJIT: on	If the value is set to "on", the miniJIT is used to compile the MIDlet suite code. If the attribute is absent or set to "off", the miniJIT is not used.
MIDlet-Push-<n>	The Push registration attribute name. Multiple push registrations can be provided in a `MIDlet` suite. The numeric value for <n> starts from 1 and *must* use consecutive ordinal numbers for additional entries. The first missing entry terminates the list. Any additional entries are ignored. Refer to "MIPD 2.0 Push Registry" on page 34 for more details.